Analysing the Control Software of the Compact Muon Solenoid Experiment at the Large Hadron Collider

Yi-Ling Hwong[1⋆], Vincent J.J. Kusters[1,2], and Tim A.C. Willemse[2]

[1] CERN, European Organization for Nuclear Research,
CH-1211 Geneva 23, Switzerland
[2] Department of Mathematics and Computer Science,
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

Abstract. The control software of the CERN Compact Muon Solenoid experiment contains over 30,000 finite state machines. These state machines are organised hierarchically: commands are sent down the hierarchy and state changes are sent upwards. The sheer size of the system makes it virtually impossible to fully understand the details of its behaviour at the macro level. This is fuelled by unclarities that already exist at the micro level. We have solved the latter problem by formally describing the finite state machines in the mCRL2 process algebra. The translation has been implemented using the *ASF+SDF meta-environment*, and its correctness was assessed by means of simulations and visualisations of individual finite state machines and through formal verification of subsystems of the control software. Based on the formalised semantics of the finite state machines, we have developed dedicated tooling for checking properties that can be verified on finite state machines in isolation.

1 Introduction

The Large Hadron Collider (LHC) experiment at the European Organization for Nuclear Research (CERN) is built in a tunnel 27 kilometres in circumference and is designed to yield head-on collisions of two proton (ion) beams of 7 TeV each. The Compact Muon Solenoid (CMS) experiment is one of the four big experiments of the LHC. It is a general purpose detector to study the wide range of particles and phenomena produced in the high-energy collisions in the LHC. The CMS experiment is made up of 7 subdetectors, with each of them designed to stop, track or measure different particles emerging from the proton collisions. Early 2010, it achieved its first successful 7 TeV collision, breaking its previous world record, setting a new one.

The control, configuration, readout and monitoring of hardware devices and the detector status, in particular various kinds of environment variables such as temperature, humidity, high voltage, and low voltage, are carried out by the Detector Control System

⋆ This work has been supported in part by a Marie Curie Initial Training Network Fellowship of the European Community's Seventh framework program under contract number (PITN-GA-2008-211801-ACEOLE).

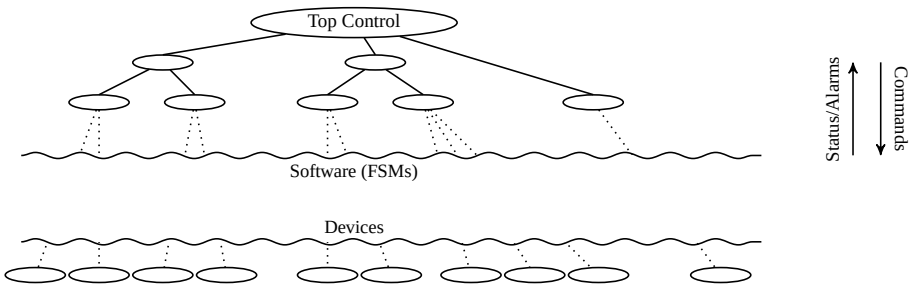

Fig. 1. Architecture of the real-time monitoring and control system of the CMS experiment, running at the LHC.

(DCS). The control software of the CMS detector is implemented with the Siemens commercial Supervision, Control And Data Acquisition (SCADA) package PVSS-II and the CERN Joint Controls Project (JCOP) framework [9]. The architecture of the control software for all four big LHC experiments is based on the SMI++ framework [5, 6]. Under the SMI++ framework, the real world is viewed as a collection of objects behaving as finite state machines (FSMs). These FSMs are described using the State Manager Language (SML). A characteristic of the used architecture is the regularity and relatively low complexity of the individual FSMs and device drivers that together constitute the control software; the main source of complexity is in the cooperation of these FSMs. Cooperation is strictly hierarchical, consisting of several layers; see Figure 1 for a schematic overview. The FSMs are organised in a tree structure where every node has one parent and zero or more children, except for the top node, which has no parent. Nodes communicate by sending commands to their children and state updates to their parents, so commands are refined and propagated down the hierarchy and status updates are sent upwards. Hardware devices are typically found only at the bottom-most layer.

The FSM system in the CMS experiment contains over 30,000 nodes. On average, each FSM contains 5 logical states. Based on our early experiments with some subsystems, we believe that $10^{30,000}$ states is a very conservative estimate of the size of the state space for the full control system. The sheer size of the system significantly contributes to its complexity. Complicating factors in understanding the behaviour of the system are the diversity in the development philosophies in subgroups responsible for controlling their own subdetectors, and the huge amount of parameters to be monitored. In view of this complexity, it is currently impossible to trace the root cause of problems when unexpected behaviours manifest themselves. A single badly designed FSM may be sufficient to lead to a livelock, resulting in non-responsive hardware devices, potentially ruining expensive and difficult experiments. Considering the scientific importance of these experiments, this justifies the use of rigorous methods for understanding and analysing the system.

Our contributions are twofold. First, we have formalised SML by mapping its language constructs onto constructs in the process algebraic language mCRL2 [7]. Second, based on our understanding of the semantics of SML, we have identified properties that

can be verified for FSMs in isolation, and for which we have developed dedicated verification tooling.

Using the ASF+SDF meta-environment [12], we have developed a prototype translation implementing our mapping of SML to mCRL2. This allowed us to quickly assess the correctness of the translation through simulation and visualisation of FSMs in isolation, and by means of formal verification of small subsystems of the control software, using the mCRL2 toolset. The feedback obtained by the verification and simulation enabled us to further improve the transformation. The use of the ASF+SDF meta-environment allowed us to repeat this cycle in quick successions, and, at the same time, maintain a formal description of the translation. Although the ASF+SDF Meta Environment development was discontinued in 2010, we chose it over similar products as ATL because we were already familiar with it and because its syntax-driven, functional approach results in very clear translation rules.

Our dedicated verification tools allow the developers at CERN to quickly perform behavioural sanity checks on their design, and use the feedback of the tools to further improve on their designs in case of any problems. Results using these tools so far are favourable: with only a fraction of the total number of FSMs inspected so far, several problems have surfaced and have been fixed.

Outline We give a cursory overview of the core of the SML language in Section 2. The mCRL2 semantics of this core are then explained in Section 3, and we briefly elaborate on the methodology we used for obtaining this semantics. Our dedicated verification tools for SML, together with some of the results obtained so far, are described in further detail in Section 4. We summarise our findings and suggestions in Section 5.

2 The State Manager Language

The finite state machines used in the CMS experiment are described in the State Manager Language (SML) [5, 6]. We present the syntax and the suggested meaning of the core of the language using snapshots of a running example; we revisit this example in our formalisation in Section 3. Note that in reality, SML is larger than presented here, but the control system is made up largely of FSMs employing these core constructs only.

Listing 1 shows part of the definition of a *class* in SML. Conceptually, this is the same kind of class known from object-oriented programming: the class is defined once, but can be instantiated many times. An instantiation is referred to as a Finite State Machine. A class consists of one or more *state clauses*; Listing 1 only shows the state clause for the `OFF` state. Intuitively, a state clause describes how the FSM should behave when it is in a particular state. Every state clause consists of a list of *when clauses* and a list of *action clauses*, either of which may be empty.

A *when clause* has two parts: a *guard* which is a Boolean expression over the states of the children of the FSM and a *referer* which describes what should happen if the guard evaluates to true. The base form of a guard is `P in_state S`, where `S` is the name of a state (or a set of state names) and `P` is a *child pattern*. A child pattern consists of two parts: the first part is either `ANY` or `ALL` and the second part is the name of a class or the literal `FwCHILDREN`. The intended meaning is straightforward:

```
class: $FWPART_$TOP$RPC_Chamber_CLASS
    state: OFF
        when ( ( $ANY$FwCHILDREN in_state ERROR ) or
               ( $ANY$FwCHILDREN in_state TRIPPED ) )  move_to ERROR

        when ( $ANY$RPC_HV in_state {RAMPING_UP,
                                     RAMPING_DOWN} ) move_to RAMPING
        when ( ( $ALL$RPC_LV in_state ON ) and
               ( $ALL$RPC_HV in_state STANDBY ) )  move_to STANDBY

        when ( ( $ALL$RPC_HV in_state ON ) and
               ( $ALL$RPC_LV in_state ON ) )  move_to ON

        when ( ( $ALL$FwCHILDREN in_state ON ) and
               ( $ALL$RPC_T in_state OK ) )  move_to ON

        action: STANDBY
            do STANDBY $ALL$RPC_HV
            do ON $ALL$RPC_LV
        action: OFF
            do OFF $ALL$FwCHILDREN
        action: ON
            do ON $ALL$FwCHILDREN
```

Listing 1: Part of the definition of the *Chamber* class in SML.

```
$ALL$FwCHILDREN in_state ON
```

means “all children are in the `ON` state”, and:

```
$ANY$RPC_HV in_state {RAMPING_UP, RAMPING_DOWN}
```

evaluates to true if “some child of class `RPC_HV` is either in state `RAMPING_UP` or state `RAMPING_DOWN`”.

A referer is either of the form `move_to S`, indicating that the finite state machine changes its state to `S`, or of the form `do A`, indicating that the action with name `A` should be executed next. If the guards of more than one *when clause* evaluate to true, the topmost enabled referer is executed. Whenever the FSM moves to a new state, it executes the *when clauses*, starting from the top *when clause*, to see if it should stay in this state (all guards are false) or if it should go to another state (some guard is true). It is therefore possible that a single `move_to` referer or statement (see below) triggers a series of state changes.

An *action clause* consists of a *name* and a list of *statements*. When an FSM receives a command while in a state `S`, it looks inside the state clause of state `S` for an *action clause* with the same name as the command and if such an *action clause* exists, it executes its statement list. If no such action exists, the command is ignored. For example, if the *Chamber* finite state machine from Listing 1 is in state `OFF` and it receives an `ON` command, it will execute the last *action clause*.

The most commonly used statement is `do C P`, which means that the command `C` is sent to all children which match the child pattern `P`. After a command is sent, the child is marked *busy*. When a child sends its new state back, this *busy* flag is removed. The `do` statement is non-blocking, *i.e.*, it does not wait for the children to respond with their new state. The child pattern always starts with `$ALL$` in this context. SML also provides `if` and `move_to` statements, as we illustrated in Listing 2.

```
action: STANDBY
    do STANDBY $ALL$RPC_HV
    do ON $ALL$RPC_LV
    if ( $ALL$RPC_LV in_state ON ) then
        do ON $ALL$RPC_HV
        do ON $ALL$RPC_LV
        if ( $ALL$RPC_HV in_state ON ) then
            do ON $ALL$RPC_HV
            move_to ON
        endif
    else
        do STANDBY $ALL$RPC_LV
        do STANDBY $ALL$RPC_HV
        do STANDBY $ALL$FwCHILDREN
    endif
```

Listing 2: An example of a more complex *action clause*.

The `move_to S` statement immediately stops execution of the *action clause* and causes the FSM to move to the `S` state. The `if G then S1 else S2 endif` statement blocks as long as there is a child, referred to in `G`, that has a busy flag. If the guard `G` evalutates to true, then `S1` is executed and otherwise `S2` is executed. The else clause is optional.

3 A Formal Semantics for SML

We use the process algebra mCRL2 [7] to formalise the semantics of programs written in SML. The formal translation of SML into mCRL2 can be found in the appendices.

Our choice for mCRL2 is motivated largely by the expressive power of the language, its rich data language rooted in the theory of Abstract Data Types, its available tool support, and our understanding of the advantages and disadvantages of mCRL2. Before we address the translation of SML to mCRL2, we briefly describe the mCRL2 language.

3.1 A Brief Overview of mCRL2

The mCRL2 language consists of two distinct parts: a *data language* for describing the data transformations and data types, and a *process language* for specifying system behaviours. For a comprehensive language tutorial, we refer to `http://mcrl2.org`.

The data language, which is rooted in the theory of *abstract data types*, includes built-in definitions for many of the commonly used data types, such as Booleans, Integers, Natural numbers, *etc.*, and allows users to specify their own data sorts. In addition, container sorts, such as *lists*, *sets* and *bags* are available.

The process specification language of mCRL2 consists of only a small number of basic operators and primitives. The language is inspired by process algebras such as ACP [1], and has both an axiomatic and an operational semantics.

A set of (parameterised) actions are used to model atomic, observable events. Processes are constructed compositionally: the non-deterministic choice between processes `p` and `q` is denoted `p+q`; their sequential composition is denoted `p.q`, and their parallel composition is denoted `p||q`. In addition, there are facilities to enforce communication between different actions and abstracting from actions.

The main feature of the process language is that processes can depend on data. For instance, `b->p<>q` denotes a conditional choice between processes `p` and `q`: if `b` evaluates to *true*, it behaves as process `p`, and otherwise as process `q`. In a similar vein, `sum d:D.p(d)` describes a (possibly infinite) choice between processes `p` with different values for variable `d`.

3.2 From SML to mCRL2

We next present our formalisation of SML in mCRL2. Every SML class is converted to an mCRL2 process definition; the behaviour of an FSM is then described by the behaviour of a process instance. Each FSM maintains a state and a pointer to the code it is currently executing. In addition, an FSM is embedded in a global tree-like configuration that identifies its parent, and its children. In order to faithfully describe the behaviour of an FSM, we therefore equip each mCRL2 process definition for a class `X` with this information as follows:

```
proc X_CLASS(self: Id, parent: Id, s: State, chs: Children,
             phase: Phase, aArgs: ActPhaseArgs)
```

Parameter `self` represents a unique identifier for a process instance, and `parent` is the identifier of `self`'s parent in the tree. Parameter `s` is used to keep track of the state of the FSM. The state information of `self`'s children is stored in `chs` of sort `Children`, which is a list of sort `Child`, a structured sort:

```
Children = List(Child);
Child = struct child(id:Id, state:State, ptype:PType, busy:Bool);
```

The above structured sort `Child` can be thought of as a named tuple; `id` represents the unique identifier of a child, `state` is the state that this child sent to `X` in its last state-update message, `ptype` maintains the FSM class of this child, and `busy` is the flag that indicates that the child is still processing the last command `X` sent to it. This flag is set after sending a message to the child, and reset when it responds with its new state. Whenever `X` receives a state-update message from one of its children, the `chs` structure is updated accordingly. This structure is used to evaluate the *when clauses* and to determine to which processes commands have to be sent.

The `phase` parameter has value `WhenPhase` if the FSM is executing the *when clauses* and `ActionPhase` otherwise; `Phase` is a simple structured sort containing these two values. The phases will be explained in detail in the following section. Finally, `aArgs` is a structure that contains information we only need in the *action phase*. It is defined as follows:

```
ActPhaseArgs = struct actArgs(cq: CommandQueue, nrf: IdList,
                              pc: Int, rsc: Bool)
```

We forego a discussion of the `nrf` and `rsc` parameters, which are solely used during an intialisation phase. The command queue `cq` contains messages that are to be sent to an FSM's children. Specifically, when executing a `do C P` statement, we add a pair with the child's id and the command `C` to `cq`, for every child matching the child pattern `P`. The command queue is subsequently emptied by sending the messages stored in `cq`.

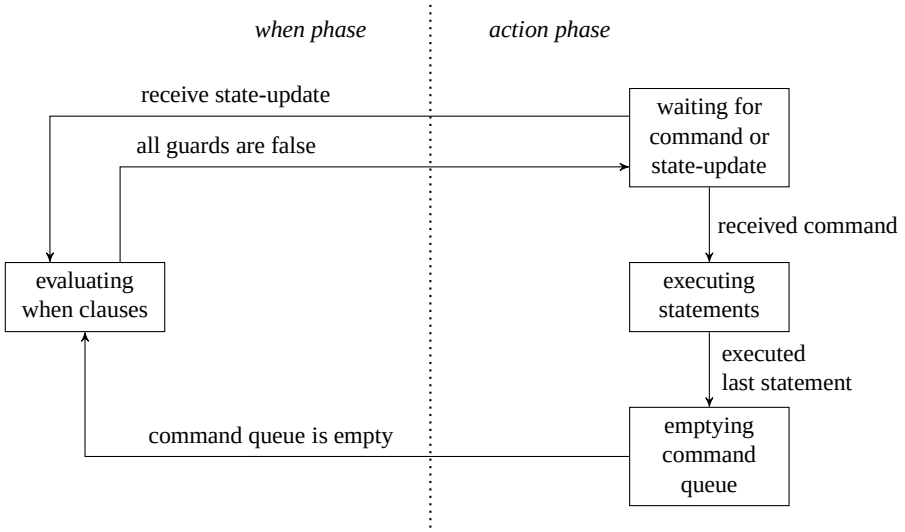

Fig. 2. Overview of the *when phase* and the *action phase*.

Phases During the *when phase*, a process executes *when clauses* until it reaches a state in which none of the guards evaluate to true. It then moves to the *action phase*. In the *action phase*, a process can receive a command from its parent or a state-update message from one of its children. This process is illustrated in Figure 2. After handling the command or message, it returns to the *when phase*.

Translating the *when phase* turns out to be rather straightforward: for each state a process term consisting of nested if-then-else statements is introduced, formalised by

mCRL2 expressions of the form `b->p<>q` (if `b`, then act as process `p`, otherwise as `q`). Each if-clause represents exactly one *when clause*. The else-clause of the last *when clause* sends a state-update message (represented by the mCRL2 action `send_state`) with the current state to the parent of this FSM and moves to the *action phase*. An example is given in Translation 1.

SML

```
state: OFF
  when G1 move_to S1
  ...
  when Gn move_to Sn
```

mCRL2

```
instate_OFF(s) && isWhenPhase(phase) -> (
  translation_of_G1 ->
    move_state(self,S1).
    X_CLASS(self,parent,S1,chs,phase,aArgs) <>
  ...
  translation_of_Gn ->
     move_state(self,Sn).
     X_CLASS(self,parent,Sn,chs,phase,aArgs) <>
  send_state(self,parent,s).
  move_phase(self,ActionPhase).
  X_CLASS(self,parent,s,chs,ActionPhase,
          reset(aArgs)))
```

Translation 1: Simplified translation of the *when clauses* of a state OFF. Note that `p.q` describes the process `p` that, upon successful termination, continues to behave as process `q`.

The `move_state` action indicates that the process changes its state. The `send_state` action communicates with the `receive_state` action to a `comm_state` action, representing the communication of the new state to the parent. Note that the state is sent only if none of the guards are true. Upon sending the new state, the process changes to the *action phase*, signalled by a `move_phase` action.

Modelling the *action phase* is more involved as we need to add some terms for initialisation and sending messages. We will focus on the translation of the *action clauses* and the code which handles state-update messages.

SML allows for an arbitrary number of statements and an arbitrary number of (nested) if-statements in every *action clause*. We uniquely identify the translation of every statement with an integer label. After executing a statement, the `pc(aArgs)` program counter is set to the label of the statement which should be executed next. There are two special cases here:

- Label 0, the clause selector. When entering the *action phase*, the program counter is set to 0. Upon receiving a command, the clause selector sets the program counter to the label of the first statement of the *action clause* that should handle the command.
- Label -1, end of action. After executing an action, the program counter is set to -1, signalling that the command queue must be emptied and the process must change to the *when phase*.

An example is given in Translation 2. The `receive_command` action models the reception of a command that was sent by the FSM's parent. Such a command is ignored if no *action clause* handles it. In the example, observe that both after ignoring a command and after completing the execution of the `STANDBY` action handler, the program counter is set to -1. A process term not shown here then empties the command queue by issueing a sequence of `send_command` actions, and subsequently returns to the *when phase*. Note that these `send_command` actions and `receive_command` actions are meant to synchronise, resulting in a `comm_command` action. This is enforced at a higher level in the specification.

SML

```
state: OFF
  action: STANDBY
    do STANDBY $ALL$Y
    do ON $ALL$Z
  action: OFF
    do OFF $ALL$Y
  action: ON
    do ON $ALL$Y
```

mCRL2

```
instate_OFF(s) && isActPhase(phase) -> (
  pc(aArgs) == 0 ->
    sum c:Command.(
      receive_command(parent,self,c).
      isC_STANDBY(c) ->
        X_CLASS(self,parent,s,chs,phase,
                update_pc(aArgs,1)) <>
      isC_OFF(c) ->
        X_CLASS(self,parent,s,chs,phase,
                update_pc(aArgs,3)) <>
      isC_ON(c) ->
        X_CLASS(self,parent,s,chs,phase,
                update_pc(aArgs,4)) <>
      send_state(self,parent,s).
      ignored_command(self,c).
      X_CLASS(self,parent,s,chs,phase,
              update_pc(aArgs,-1))) +

  pc(aArgs) == 1 ->
    RPC_Chamber_CLASS(self,parent,s,chs,phase,
      add_HV_STANDBY_commands(
        update_pc(aArgs,2))) +

  pc(aArgs) == 2 ->
    RPC_Chamber_CLASS(self,parent,s,chs,phase,
      add_LV_ON_commands(
        update_pc(aArgs,-1)) + ...
```

Translation 2: Simplified translation of the *action clauses* of a state OFF.

Since a `do` statement is asynchronous, the children can send their state-update at any time during the *action phase*. This is dealt with as follows. Suppose a state-update message is received. If this precedes the reception of a command in this *action phase*,

we simply process the state-update and move to the *when phase*. If we are in the middle of executing an *action clause*, we process the state-update, but do not move to the *when clause*.

3.3 Validating the Formalisation of SML

The challenge in formalising SML is in correctly interpreting its language constructs. We combined two strategies for assessing and improving the correctness of our semantics: informal discussions with the development team of the language and applying formal analysis techniques on sample FSMs taken from the control software.

The discussions with the SML development team were used to solidify our initial understanding of SML and its main constructs. Based on these discussions, we manually translated several FSMs into mCRL2, and validated the resulting processes manually using the available simulation and visualisation tools of mCRL2. This revealed a few minor issues with our understanding of the semantics of SML, alongside many issues that could be traced back to sloppiness in applying the translation from SML to mCRL2 manually.

In response to the latter problem, we eliminated the need for manually translating FSMs to mCRL2. To this end, we utilised the ASF+SDF meta-environment (see [12, 10]) to rapidly prototype an automatic translator that, ultimately, came to implement the translation scheme we described in the previous section. The *Syntax Definition Formalism* (SDF) was used to describe the syntax of both SML and mCRL2, whereas the *Algebraic Specification Formalism* (ASF) was used to express the term rewrite rules that are needed to do the actual translation. Apart from the gains in speed and the consistency in applying the transformations that were brought about by the automation, the automation also served the purpose of formalising the semantics of SML.

The final details of our semantics were tested by analysing relatively well-understood subsystems of the control software in mCRL2. We briefly discuss our findings using a partly simplified subsystem, colloquially known as the *Wheel*, see Figure 3. The Wheel subsystem is a component of the Resistive Plate Chamber (RPC) subdetector of the CMS experiment. It belongs to the barrel region of the RPC subdetector. Each Wheel subsystem contains 12 sectors, each sector is equipped with 4 muon stations which are made of Drift Tube chambers. We forego a detailed formal discussion of this subsystem (for details, we refer to [11]), but only address our analysis of this subsystem using formal analyses techniques, and the impact this had on our understanding of the semantics and the transformation. It is important to keep in mind that the analysis was conducted primarily to assess the quality of our translation, the correctness of the subsystem being only secondary.

The mCRL2 specification of the *Wheel* subsystem was obtained by combining the mCRL2 processes obtained by running our prototype implementation on each involved FSM. Generating the state space of the *Wheel* subsystem takes roughly one minute using the symbolic state space generation tools offered by the LTSmin tools [4]. This toolset can be integrated in the mCRL2 toolset. For the discussed configuration, the state space is still of modest proportions, measuring slightly less than 5 million states and 24 million transitions. Varying the amount of children of class *Sector* causes a dramatic growth of the state space. Using 3 instead of 2 children of class *Sector* yields

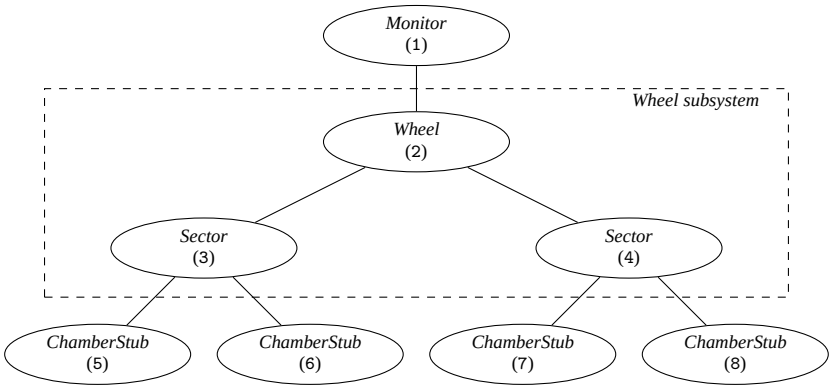

Fig. 3. A schematic overview of our model of the *Wheel* subsystem, and its used FSMs. The identifiers of the processes representing the FSMs are given between parentheses; these were used in our analyses.

roughly 800 million states; using 4 children of class *Sector*, leads to 120 billion states, and requires half a day.

Apart from repeating the simulations and visualisations, at this stage we also applied *model checking* to systematically probe the translation. Together with the development team of the *Wheel* subsystem, a few basic requirements were formalised in the first-order modal μ-calculus [8], see Table 1. The first-order modal μ-calculus is the default requirement specification language in the mCRL2 toolset.

The studied subsystem was considered to satisfy all stated properties. While smoothing out details in the translation of SML to mCRL2, the deadlock-freedom property was violated every now and then, indicating issues with our interpretation of SML. These were mostly concerned with the semantics of the blocking and non-blocking constructs of SML, and the complex constructs used to model the message passing between FSMs and their children.

The absence of intermediate states in the *when phase* was violated only once in our verification efforts. A more detailed scrutiny of the run revealed a problem in our translation, which was subsequently fixed.

The third requirement, stating the inevitability of a state change by a child once such a state change has been commissioned, failed to hold. The violation is caused by the overriding of commands by subsequent commands that are issued immediately. Discussions with the development teams revealed that the violations are real, *i.e.*, they are within the range of real behaviour, suggesting that our formalisation was adequate. The property was modified to ignore the spurious runs, resulting in the following property:

```
nu X. [true]X &&
    [comm_command(i,i_c,c)](mu Y. <true>true &&
        [!(comm_state(i_c,i,c2s(c)) ||
           exists c':Command. comm_command(i,i_c,c'))]Y)
```

The final requirement also failed to hold. The violation is similar spirited to the violation of the third requirement, and, again found to comply to reality. The weakened

Table 1. Basic requirements for the *Wheel* subsystem; `i:Id` denotes an identifier of an FSM; `i_c:Id` denotes a child of FSM `i`; `c:Command` denotes a command;`c2s(c)` denotes the state with the homonymous command name, *e.g.*, `c2s(ON) = ON`.

1. Absence of deadlock:
```
nu X. [true]X && <true>true
```
2. Absence of intermediate states in the *when phase*:
```
nu X. [true]X &&
     [exists s:State. move_state(i,s)](nu Y.
         [(!move_phase(i,ActionPhase))]Y
      && [exists s:State. move_state(i,s)]false)
```
3. Responsiveness:
```
nu X. [true]X &&
     [comm_command(i,i_c,c)](mu Y.
         <true>true && [!comm_state(i_c,i,c2s(c))]Y)
```
4. Progress:
```
nu X. [true]X &&
     mu Y. <exists s:State. move_state(i,s)>true
||
          (<true>true && [true]Y)
```

requirement that was subsequently agreed upon expresses the attainability of some state change:

```
nu X. [true]X &&
    mu Y. <exists s:State. move_state(i,s)>true || <true>Y
```

Neither visual inspection of the state space using 2D and 3D visualisation tools, nor simulation using the mCRL2 simulators revealed any further incongruences in our final formalisation of SML, sketched in the previous section.

4 Dedicated Tooling for Verification

Some desired properties, such as the absence of loops within the *when phase*, can be checked by analysing an FSM in isolation, using the transformation to mCRL2. However, the verifications using the modal μ-calculus currently require too much overhead to serve as a basis for lightweight tooling that can be integrated in the SML development environment.

In an attempt to improve on this situation, we explored the possibilities of using *Bounded Model Checking* (BMC) [3, 2]. The basic idea of BMC is to check for a counterexample in bounded runs. If no bugs are found using the current bound, then the bound is increased until either a bug is found, the problem becomes intractable, or some pre-determined upper bound is reached upon which the verification is complete.

The BMC problem can be efficiently reduced to a propositional satisfiability problem, and can therefore be solved by SAT methods. SAT procedures do not necessarily suffer from the space explosion problem, and a modern SAT solver can handle formulas with hundreds of thousands of variables or more, see *e.g.* [2].

We have applied BMC techniques for the detection of `move_to` loops and the detection of unreachable states and trap states. As an example of a `move_to` loop, consider the excerpt of the `ECALfw_CoolingDee` FSM class in Listing 3, which our tool found to contain issues. If an instance of `ECALfw_CoolingDee` has one child in state `ERROR` and one in state `NO_CONNECTION`, it will loop indefinitely between these two states. Once this happens, an entire subsystem may enter a livelock and become unresponsive.

```
state: ERROR
  when ( $ANY$FwCHILDREN in_state NO_CONNECTION ) move_to NO_CONNECTION
  when ( $ALL$FwCHILDREN in_state OK ) move_to OK

state: NO_CONNECTION
  when ( $ALL$FwCHILDREN in_state OK ) move_to OK
  when ( $ANY$FwCHILDREN in_state ERROR ) move_to ERROR
```

Listing 3: An excerpt from the `ECALfw_CoolingDee` FSM that exhibits a loop within the *when phase*.

We first convert this problem into a graph problem as follows. Let $\mathcal{F}$ be an FSM and $\mathcal{M}$ be a Kripke structure. A state in $\mathcal{M}$ corresponds to the combined state of $\mathcal{F}$ and its children, *e.g.*, if $\mathcal{F}$ is in state `ON` and has two children which are in state `OFF`, then the corresponding state in $\mathcal{M}$ is $(\texttt{ON}, \texttt{OFF}, \texttt{OFF})$. There is a transition between two states s_1 and s_2 in $\mathcal{M}$ if and only if s_1 can do a `move_to` action to s_2 in $\mathcal{F}$. Moreover, every state in $\mathcal{M}$ is an initial state. It thus suffices to inspect $\mathcal{M}$ instead of $\mathcal{F}$, as stated by the following lemma:

Lemma 1. *$\mathcal{F}$ contains a loop of `move_to` actions if and only if $\mathcal{M}$ contains a loop.*

We next translate the problem of detecting a loop in $\mathcal{M}$ into a SAT problem. First, we consider executions of length k; afterwards, we show that we can statically choose k such that we can find every loop.

Let the predicate in_state be defined as follows: $in\_state(s, p, i)$ holds if and only if the process with identifier p is in state s after i steps. We assign the identifier zero to the FSM under consideration and the numbers $1, 2, 3, \ldots$ to its children. The resulting formula will have three components: the *state constraints*, the *transition relation* and the *loop condition*.

Using the state constraints, we ensure the FSM to always be in exactly one state. Moreover, the states of the children should not change during the execution of the *when phase*, per the semantics in the previous section. This is straigthforwardly expressed as a boolean formula on the in_state predicate.

Next, we encode the transition relation: the relation between $in\_state(s, 0, i)$ and $in\_state(s', 0, i+1)$ for every i. In other words: the `move_to` steps the parent process is

allowed to take. This involves converting the *when clauses* for each state of the parent FSM, taking care the semantics as outlined in the previous section is reflected. The last ingredient is the loop condition: if $in\_state(s, 0, 0)$ holds, then $in\_state(s, 0, i)$ must hold for some $i > 1$, indicating that the parent returned to the state in which it started.

The final SAT formula is obtained by taking the conjunction of the state constraints, the transition relation and the loop condition. It is not hard to see that if this formula is satisfiable, then there is a loop in $\mathcal{M}$ and hence in $\mathcal{F}$. It is more difficult to show that if there is a loop, then the formula is satisfiable. Let n be the total number of states of the FSM and let n_t be the total number of states of each child class t. We then have the following result:

Theorem 1. *All possible loops in $\mathcal{F}$ can be found by considering paths of length at most n in an FSM configuration $\mathcal{F}$ having n_t children for each child class t.*

Proof (sketch). Since $\mathcal{F}$ only has n states, the longest possible loop also contains n states. Since every state in $\mathcal{M}$ is an initial state, every possible loop can by found by doing n steps from an initial state.

It remains to show that all loops can be found by considering a configuration with n_t children for each child class t. This follows from the fact that SML guards are restricted to check for *any* or *all* children in a particular state. □

A second desirable behavioural property of an FSM is that all states should remain reachable during the execution of an FSM. While we can again easily encode this property into the modal μ-calculus, we use a more direct approach to detect violations of this property by constructing a graph that captures all potential state changes. For this, we determine whether there is a configuration of children such that $\mathcal{F}$ can execute a `move_to` action from a state s to a state s'. Doing so for all pairs (s, s') of states of $\mathcal{F}$ yields a graph encoding all possible state changes of $\mathcal{F}$.

Computing the strongly connected components (SCCs) of the thusly obtained graph gives sufficient information to pinpoint violations to the reachability property: the presence of more than a single SCC means that one cannot move back and forth these SCCs (by definition of an SCC), and, therefore, their states. Note that this is an under-approximation of all errors that can potentially exist, as the actual reachability dynamically depends on the configuration of the children of an FSM. Still, as the state change graph of the `ESfw_Endcap` FSM class in Figure 4 illustrates, issues can be found in production FSMs: the `OFF` state can never be reached from any of the other states. Using the graphs generated by our tools, such issues are quickly explained and located.

Results The results using our dedicated tools for performing these behavioural sanity checks on isolated FSMs are very satisfactory: of the several hundreds of FSM classes contained in the control system, we so far analysed 40 FSM classes and found 6 to contain issues. In 4 of these, we found logical errors that could give rise to livelocks in the system due to the presence of loops in the *when phase*; an example thereof is given in Listing 3. Somewhat unexpectedly, all loops were found to involve two states. Note that the size of the average FSM class (in general more than 100 lines of SML code, and at least two children) means that even short loops such as the ones identified so far remain unnoticed and are hard to pinpoint. The remaining two FSM classes were

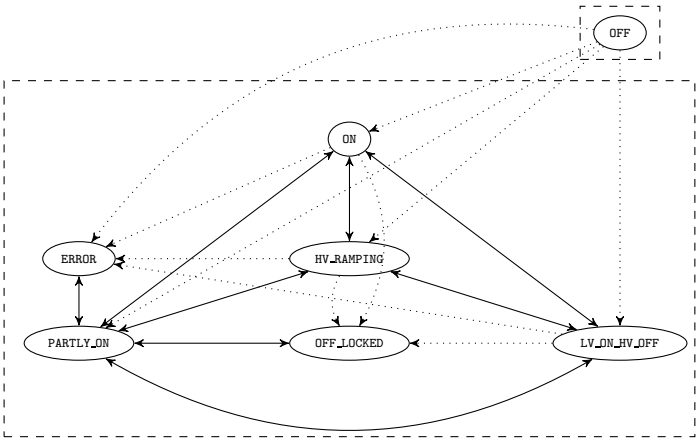

Fig. 4. The state change graph for the `ESfw_Endcap` FSM class. The solid lines are bidirectional; the dotted lines are unidirectional state changes. The SCCs are indicated by the dashed frames.

found to violate the required reachability of states, see *e.g.* Figure 4. The speed at which the errors can be found (generally requiring less than a second) means that the sanity checks could easily be incorporated in the design cycle of the FSMs.

5 Conclusion

We discussed and studied the State Machine Language (SML) that is currently used for programming the control software of the CMS experiment running at the Large Hadron Collider. To fully understand the language, we formalised it using the process algebraic language mCRL2. The quality of our formalisation was assessed using a combination of simulation and visualisation of the behaviour of FSMs in isolation and formally verifying small subsystems using model checking. To facilitate, among others, the assessment, the translation of SML to mCRL2 was implemented using the ASF+SDF meta-environment. Based on our understanding of the semantics of SML, we have built dedicated tools for performing sanity checks on isolated FSMs. Using these tools we found several issues in the control system. These tools have been well-received by the engineers at CERN, and are considered for inclusion in the development environment.

Our formalisation of SML opens up the possibility of verifying realistically large subsystems of the control system; clearly, it will be one of the most challenging verification problems currently available. In our analysis of the *Wheel* subsystem, we have only used a modest set of tools for manipulating the state space; symmetry reduction, partial order reduction, parallel exploration techniques, abstractions and abstract interpretation were not considered at this point. It remains to be investigated how such techniques fare on this problem.

Acknowledgments. We thank Giel Oerlemans, Dennis Schunselaar and Frank Staals from the Eindhoven University of Technology for their contribution to a first draft of the

ASF+SDF translation. We also thank Frank Glege and Robert Gomez-Reino Garrido from the CERN CMS DAQ group for their support and advice, and Clara Gaspar for discussions on SML. Jaco van de Pol is thanked for his help with the LTSmin toolset.

References

1. J.C.M. Baeten, T. Basten, and M.A. Reniers. *Process Algebra: Equational Theories of Communicating Processes*, volume 50 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2010.
2. A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. *Advances in Computers*, 58:118–149, 2003.
3. A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proc. of TACAS*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.
4. Stefan Blom, Jaco van de Pol, and Michael Weber 0002. LTSmin: Distributed and symbolic reachability. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *LNCS*, pages 354–359. Springer, 2010.
5. B. Franek and C. Gaspar. SMI++ object-oriented framework for designing and implementing distributed control systems. *IEEE Transactions on Nuclear Science*, 52(4):891–895, 2005.
6. C. Gaspar and B. Franek. SMI++—Object-oriented framework for designing control systems for HEP experiments. *Computer physics communications*, 110(1–3):87–90, 1998.
7. J.F. Groote, A.H.J. Mathijssen, M.A. Reniers, Y.S. Usenko, and M.J. v. Weerdenburg. Analysis of distributed systems with mCRL2. In *Process Algebra for Parallel and Distributed Processing*, pages 99–128. Chapman Hall, 2009.
8. J.F. Groote and T.A.C. Willemse. Model-checking processes with data. *Science of Computer Programming*, 56(3):251–273, 2005.
9. O. Holme, M. González-Berges, P. Golonka, and S. Schmeling. The JCOP Framework. Technical Report CERN-OPEN-2005-027, CERN, Geneva, Sep 2005.
10. P. Klint. A meta-environment for generating programming environments. *ACM Trans. Softw. Eng. Methodol.*, 2(2):176–201, 1993.
11. P. Paolucci and G. Polese. The detector control systems for the cms resistive plate chamber, 2008. CERN-CMS-NOTE-2008-036, see `http://cdsweb.cern.ch/record/1167905`.
12. M. Van den Brand, A. Van Deursen, J. Heering, H.A. De Jong, M. De Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. The ASF+SDF meta-environment: A component-based language development environment. In Reinhard Wilhelm, editor, *Proc. of Compiler Construction*, volume 2027 of *LNCS*, pages 365–370. Springer, 2001.

A ASF and SDF files

A.1 midtools.sdf

```
%% Module that defines the identifiers used throughout our languages.

module midtools
imports
  basic/Whitespace
  basic/Comments
  basic/Booleans
  basic/Integers

exports
sorts
  %% B3 from the mCRL2 spec.
  MId
  MIds

lexical restrictions
  MId -/- [a-zA-Z0-9\_\']

lexical syntax
  [a-zA-Z\_] ([a-zA-Z0-9\_\'])* -> MId

context-free syntax
  %% Identifiers (B3)
  {MId ","}+ -> MIds

  %% Concatenate two MIds.
  concat(MId, MId)                                          -> MId

  %% Return whether an MId is in a list of MIds.
  contains(MId, MId*)                                       -> Boolean

  %% Given a list, remove all duplicates. If the list contains two identical elements,
  %% only the *rightmost* element will be preserved.
  removeDuplicates(MId*)                                    -> MId*

  %% Remove all occurrences of an element from a list.
  remove(MId, MId*)                                         -> MId*

  %% Compute the set intersection of two lists. The resulting list has no duplicates.
  intersect(MId*, MId*)                                     -> MId*

  %% Returns true iff the list is empty.
  empty(MId*)                                               -> Boolean

  %% Returns the length of a list of MIds.
  length(MId*)                                              -> Integer

hiddens
variables
  "$mid"[0-9]*                    -> MId
  "$mid+"[0-9]*                   -> MId+
  "$mid*"[0-9]*                   -> MId*
  "$i"                            -> Integer

lexical variables
  "#midHead"[0-9]*                -> [a-zA-Z\_]
  "#midTailChar"[0-9]*            -> ([a-zA-Z0-9\_\'])
  "#midTail"[0-9]*                -> ([a-zA-Z0-9\_\'])*
```

A.2 midtools.asf

```
equations

[concat-1]
concat(mid(#midHead1 #midTail1), mid(#midHead2 #midTail2)) = mid(#midHead1 #midTail1 #midHead2 #midTail2)

[contains-empty]
contains($mid, ) = false

[contains-match]
contains($mid, $mid $mid*) = true

[contains-nomatch]
$mid1 != $mid2
===>
contains($mid1, $mid2 $mid*) = contains($mid1, $mid*)
```

```
[removeDuplicates-empty]
removeDuplicates() =

[removeDuplicates-many-nonunique]
contains($mid, $mid*) == true
===>
removeDuplicates($mid $mid*) = removeDuplicates($mid*)

[removeDuplicates-many-unique]
contains($mid, $mid*) == false
===>
removeDuplicates($mid $mid*) = $mid removeDuplicates($mid*)

[remove-empty]
remove($mid, ) =

[remove-many-nomatch]
$mid1 != $mid2
===>
remove($mid1, $mid2 $mid*) = $mid2 remove($mid1, $mid*)

[remove-many-match]
remove($mid,  $mid $mid*) = remove($mid, $mid*)

[intersect-empty]
intersect(, $mid*2) =

[intersect-many-match]
contains($mid, $mid*2) == true
===>
intersect($mid $mid*1, $mid*2) = $mid intersect($mid*1, $mid*2)

[intersect-many-nomatch]
contains($mid, $mid*2) == false
===>
intersect($mid $mid*1, $mid*2) = intersect($mid*1, $mid*2)

[empty-true]
empty() = true

[empty-false]
empty($mid+) = false

[length-empty]
length() = 0

[length-many]
$i := length($mid*)
===>
length($mid $mid*) = $i + 1
```

A.3 cfsm.sdf

```
%% Authors:
%%  Vincent Kusters
%%  Dennis Schunselaar

module cfsm
imports
  basic/Comments
  basic/Whitespace
  midtools

exports
context-free start-symbols
  FSMSpecification


sorts
  Identifier
  FSMSpecification
  FSMClass
  FSMStateClause
  FSMWhenClause
  FSMReferer
  FSMActionClause
  FSMStatement
  FSMParameter

  FSMExpression

  FSMChildrenSpec
  FSMChildrenAny
```

```
  FSMChildrenAll
  FSMChildrenAnySpecific
  FSMChildrenAnyFwChildren
  FSMChildrenAllSpecific
  FSMChildrenAllFwChildren

  FSMClassName
  FSMStateName
  FSMStateNameSpec
  FSMActionName


lexical syntax
  "!" "=" ~[\n]* [\n] -> LAYOUT
  "/associated" ~[\n]* [\n] -> LAYOUT
  "/ASSOCIATED" ~[\n]* [\n] -> LAYOUT
  [A-zA-ZO-9]* -> Identifier

context-free syntax
  %% Rule for the top level sort.
  FSMClass+                                                -> FSMSpecification

  %% Rules for the various clauses.
  "class: $FWPART_$TOP$" FSMClassName FSMStateClause+      -> FSMClass
  "state:" FSMStateName FSMWhenClause* FSMActionClause*    -> FSMStateClause
  "when" "(" FSMExpression ")" FSMReferer+                 -> FSMWhenClause
%%  "when" "(" "not"  "(" FSMExpression ")" ")" FSMReferer+  -> FSMWhenClause
  "action:" FSMActionName FSMStatement*                    -> FSMActionClause

  %% Rules for the statements.
   Identifier                                        -> FSMParameter
  "move_to" FSMStateName                             -> FSMStatement
  "do" FSMActionName FSMChildrenSpec                 -> FSMStatement
  "do" FSMActionName "(" FSMParameter "=" FSMParameter ")" FSMChildrenSpec -> FSMStatement
  "if" "(" FSMExpression ")" "then" FSMStatement+ ("else" FSMStatement+)? "endif" -> FSMStatement

  %% Rules for the referers.
  "move_to" FSMStateName                             -> FSMReferer
  "do" FSMActionName                                 -> FSMReferer

  %% Rules for expressions.
  FSMChildrenSpec "in_state"     FSMStateNameSpec      -> FSMExpression
  FSMChildrenSpec "not_in_state" FSMStateNameSpec      -> FSMExpression
  "not" "(" FSMExpression ")"                          -> FSMExpression
  "not" "(" FSMExpression ")" "and" "(" FSMExpression ")" -> FSMExpression


  "(" FSMExpression ")"                              -> FSMExpression
  FSMExpression "and" FSMExpression                  -> FSMExpression {left}
  FSMExpression "or"  FSMExpression                  -> FSMExpression {left}

  %% Rules for state name specifications.
  FSMStateName                                       -> FSMStateNameSpec
  "{" {FSMStateName ","}* "}"                        -> FSMStateNameSpec

  %% Rules for sets of children.
  "(" FSMChildrenSpec ")"  -> FSMChildrenSpec
  FSMChildrenAny           -> FSMChildrenSpec
  FSMChildrenAll           -> FSMChildrenSpec

  FSMChildrenAnySpecific   -> FSMChildrenAny
  FSMChildrenAnyFwChildren -> FSMChildrenAny
  FSMChildrenAllSpecific   -> FSMChildrenAll
  FSMChildrenAllFwChildren -> FSMChildrenAll

  "$ANY$" FSMClassName     -> FSMChildrenAnySpecific
  "$ANY$FwCHILDREN"        -> FSMChildrenAnyFwChildren
  "$ANY$FwCHILDREN"        -> FSMChildrenAnySpecific {reject}

  "$ALL$" FSMClassName     -> FSMChildrenAllSpecific
  "$ALL$FwCHILDREN"        -> FSMChildrenAllFwChildren
  "$ALL$FwCHILDREN"        -> FSMChildrenAllSpecific {reject}

  MId -> FSMClassName
  MId -> FSMStateName
  MId -> FSMActionName
```

A.4 mcrlt.sdf

```
%% Author: Vincent Kusters.

module mcrlt
imports
```

```
  basic/Whitespace
  basic/Comments
  basic/Integers
  midtools

exports
sorts
  %%Number

  %% B4
  SortExpr
  Domain
  SortSpec
  SortDecl
  ConstrDecl
  ProjDecl
  ProjDecls

  %% B5
  IdDecl
  IdsDecl
  OpSpec
  OpDecl

  %% B6
  EqnSpec
  EqnDecl

  %% B7
  DataExpr
  DataExprs
  BagEnumElt
  BagEnumElts

  %% B8
  MAId
  MAIdSet
  CommExpr
  CommExprSet
  RenExpr
  RenExprSet

  %% B9
  ProcExpr

  %% B10
  ActDecl
  ActSpec

  %% B11
  ProcDecl
  ProcSpec
  Init

  %% B12
  MCRL2Spec

lexical restrictions
  MId -/- [a-zA-Z0-9\_\']

context-free start-symbols
  MCRL2Spec

lexical syntax
  %% We disallow comments starting with "%%", since this leads to
  %% ambiguity with ASF+SDF comments.
  [\r\t\n\ ]       -> LAYOUT
  "%" (~[\%] ~[\n]*)? [\n]  -> LAYOUT

  %% Identifiers (B3)
  %% We will use Integer instead of Number.

context-free syntax
  %% Keywords (B1)
  "sort"    -> MId {reject}
  "cons"    -> MId {reject}
  "map"     -> MId {reject}
  "var"     -> MId {reject}
  "eqn"     -> MId {reject}
  "act"     -> MId {reject}
  "proc"    -> MId {reject}
  "init"    -> MId {reject}
  "delta"   -> MId {reject}
  "tau"     -> MId {reject}
  "sum"     -> MId {reject}
  "block"   -> MId {reject}
```

```
"allow"   -> MId {reject}
"hide"    -> MId {reject}
"rename"  -> MId {reject}
"comm"    -> MId {reject}
"struct"  -> MId {reject}
"Bool"    -> MId {reject}
"Pos"     -> MId {reject}
"Nat"     -> MId {reject}
"Int"     -> MId {reject}
"Real"    -> MId {reject}
"List"    -> MId {reject}
"Set"     -> MId {reject}
"Bag"     -> MId {reject}
"true"    -> MId {reject}
"false"   -> MId {reject}
"whr"     -> MId {reject}
"end"     -> MId {reject}
"lambda"  -> MId {reject}
"forall"  -> MId {reject}
"exists"  -> MId {reject}
"div"     -> MId {reject}
"mod"     -> MId {reject}
"in"      -> MId {reject}

%% Sort expressions and sort declarations (B4)
"Bool"                                -> SortExpr
"Pos" | "Nat" | "Int" | "Real"        -> SortExpr
"List" "(" SortExpr ")"               -> SortExpr
"Set" "(" SortExpr ")"                -> SortExpr
"Bag" "(" SortExpr ")"                -> SortExpr
MId                                   -> SortExpr
"(" SortExpr ")"                      -> SortExpr
Domain "->" SortExpr                  -> SortExpr

{SortExpr "#"}+                        -> Domain
"sort" SortDecl+                       -> SortSpec
MIds ";"                               -> SortDecl
MIds "=" SortExpr ";"                  -> SortDecl
MIds "=" "struct" {ConstrDecl "|"}+ ";" -> SortDecl

%% Difference with the specification in the reader:
%% the "? MId" part is obligatory.
MId ("(" ProjDecls ")")? ("?" MId)      -> ConstrDecl
(MId ":")? Domain                       -> ProjDecl
{ProjDecl ","}+                         -> ProjDecls

%% Declarations of constructors and mappings (B5)
MId ":" SortExpr          -> IdDecl
MIds ":" SortExpr ";"?    -> IdsDecl
("cons" | "map") OpDecl+  -> OpSpec
IdsDecl ";"               -> OpDecl

%% Declaration of equations (B6)
"eqn" EqnDecl+                          -> EqnSpec
"var" IdsDecl+ "eqn" EqnDecl+           -> EqnSpec
DataExpr "=" DataExpr ";"               -> EqnDecl
DataExpr "->" DataExpr "=" DataExpr ";" -> EqnDecl

%% Data expressions (B7)
MId | Integer | "true" | "false" | "[]" | "{}" -> DataExpr
"[" DataExprs "]"                               -> DataExpr
"{" DataExprs "}"                               -> DataExpr
"{" BagEnumElts "}"                             -> DataExpr
"{" IdDecl "|" DataExpr "}"                     -> DataExpr
"(" DataExpr ")"                                -> DataExpr
%% DataExpr with arguments was moved to the context-free priority section.
("!" | "#") DataExpr                            -> DataExpr
("forall" | "exists") IdDecl "." DataExpr       -> DataExpr
%% DataExpr with the binary operators was moved to the context-free priority
%% section.
"lambda" IdDecl "." DataExpr                    -> DataExpr
DataExpr "whr" DataExprs "end"                  -> DataExpr

{DataExpr ","}+                                 -> DataExprs
DataExpr ":" DataExpr                           -> BagEnumElt
{BagEnumElt ","}*                               -> BagEnumElts

%% Communication and renaming (B8)
{MId "|"}+                -> MAId
"{" {MAId ","}* "}"       -> MAIdSet
MAId ("->" MId)?          -> CommExpr
"{" {CommExpr ","}* "}"   -> CommExprSet
MId "->" MId              -> RenExpr
"{" {RenExpr ","}* "}"    -> RenExprSet
```

```
  %% Process expressions (B9)
  MId                                                   -> ProcExpr
  MId "(" DataExprs ")"                                 -> ProcExpr
  "delta"                                               -> ProcExpr
  "tau"                                                 -> ProcExpr
  %% Sum was moved to the context-free priorities section.
  ("block" | "allow" | "hide") "(" MAIdSet "," ProcExpr ")"  -> ProcExpr
  "rename" "(" RenExprSet "," ProcExpr ")"              -> ProcExpr
  "comm" "(" CommExprSet "," ProcExpr ")"               -> ProcExpr
  "(" ProcExpr ")"                                      -> ProcExpr
  %% ProcExpr with binary operators was moved to the context-free
  %% priorities section.

  %% Action declaration (B10)
  MIds (":" Domain)? ";"  -> ActDecl
  "act" ActDecl+          -> ActSpec

  %% Process and initial state declaration (B11)
  MId "=" ProcExpr ";"                          -> ProcDecl
  MId "(" {IdsDecl ","}+ ")" "=" ProcExpr ";"   -> ProcDecl
  "proc" ProcDecl+                              -> ProcSpec
  "init" ProcExpr ";"                           -> Init

  %% Syntax of an mCRL2 specification (B12)
  SortSpec* OpSpec* EqnSpec* ActSpec* ProcSpec* Init* -> MCRL2Spec

context-free priorities
  %% B9
  ProcExpr "@" ProcExpr                 -> ProcExpr         >
  "sum" {IdDecl ","}+ "." ProcExpr      -> ProcExpr         >
  ProcExpr "." ProcExpr                 -> ProcExpr {right} >
  ProcExpr "<<" ProcExpr                -> ProcExpr {left}  >
  {
    ProcExpr "||"  ProcExpr             -> ProcExpr {right}
    ProcExpr "|"   ProcExpr             -> ProcExpr {right}
    ProcExpr "||_" ProcExpr             -> ProcExpr {right}
  }                                                         >
  DataExpr "->" ProcExpr "<>" ProcExpr  -> ProcExpr         >
  DataExpr "->" ProcExpr                -> ProcExpr         >
  ProcExpr "+" ProcExpr                 -> ProcExpr {right} >

  %% B7
  DataExpr "(" DataExprs ")"                                  -> DataExpr        >
  DataExpr ("|>" | "<|") DataExpr                             -> DataExpr {left} >
  DataExpr "++" DataExpr                                      -> DataExpr {left} >
  DataExpr ("." | "*" | "div") DataExpr                       -> DataExpr {left} >
  DataExpr ("+" | "-") DataExpr                               -> DataExpr {left} >
  DataExpr ("mod"  | "in") DataExpr                           -> DataExpr {left} >

  DataExpr ("==" | "!=" | "<" | ">" | "<=" | ">=") DataExpr   -> DataExpr {left} >
  DataExpr ("&&" | "||" | "=>") DataExpr                      -> DataExpr {left}
```

A.5 cfsm2mcrl2.sdf

```
%% Module for converting the CERN Finite State Machines into mCRL2 code.

module cfsm2mcrl2

imports cfsm
imports mcrlt
imports genericclauses
imports midtools
imports basic/Integers
imports basic/BoolCon

exports

context-free start-symbols
SortDecl+ ProcExpr

context-free syntax

  %% Main convertor function to convert a FSM class to an MCRL2 specification.
  cfsm2mcrl2(FSMClass+) -> ProcSpec+

  %% Convertor function to convert a FSM class to an MCRL2 specfication, with the added
  %% property that the result will be a bottom monitor. That is, it has no children and
  %% whenever it would normally check the state of children, it will instead check
  %% randomStateChanges.
  cfsm2mcrl2bm(FSMClass+) -> ProcSpec+

  %% Function to generate the PType, State and Command sorts from a number of FSM classes.
```

```
  fsmGenerateSorts(FSMClass+) -> SortDecl+

  %% Function to generate the list of process names from a list of process specifications.
  mcrl2GetPTypes(ProcSpec+) -> SortDecl
hiddens

sorts
  ProcName ActionClauseTuple UniqueProcName PC

context-free syntax

  %%%%%%%%%%%% Specification Conversion
  MId                                                    -> ProcName
  Integer                                                -> PC
  <FSMActionName, PC, DataExpr, ProcExpr>                -> ActionClauseTuple

  %% Function to convert a number of FSM classes into process definitions.
  fsmClasses2Mcrl2Procs(FSMClass+, BoolCon) -> ProcSpec+

  %% The main converion function to convert one FSM Class into a Process.
  fsmClass2Mcrl2Proc(FSMClass, BoolCon)     -> ProcSpec
  fsmClassName2ProcName(MId, BoolCon)       -> MId

  %% Conversion functions for a list of states and a single state in the FSM.
  convertStates(FSMStateClause*, ProcName, BoolCon, MId*)       -> ProcExpr
  convertState(FSMStateClause, ProcName, BoolCon, MId*)         -> ProcExpr

  %% Conversion functions for the parts in each state. These functions are
  %% grouped by phase. First the functions needed in the when-phase. Functions
  %% that are requird in both phases are listed in with the when-phase.

  %% The conversion of the when-clauses requires a third parameter: the name of
  %% the state we are currently converting.
  convertWhenClauses(FSMWhenClause*, ProcName, MId, ActionClauseTuple*) -> ProcExpr
  convertReferer(FSMReferer, ProcName, MId, ActionClauseTuple*)         -> ProcExpr
  convertExpr(FSMExpression)                                            -> DataExpr
  convertChildrenSpec(FSMChildrenSpec)                                  -> DataExpr
  convertStateNameSpec(FSMStateNameSpec)                                -> DataExpr

  %% Conversion functions for the action clauses.
  combineActionClauseComponents(ActionClauseTuple*, ProcName, FSMStateName) -> ProcExpr

  %% Helper function for convertActionClauseComponents and convertReferer.
  %% Returns the fsm action name, mcrl2 condition and mcrl2 effect of an fsm
  %% action clause. Action clauses also need the name of the state. As with the
  %% when-clauses, this is the third parameter.
  gatherComponentsFromActionClauses(FSMActionClause*, ProcName, MId, PC) -> <ActionClauseTuple*, PC>

  %% Helper function for convertActionClauseComponents.
  getActionClauseTupleForActionName(ActionClauseTuple*, FSMActionName) -> ActionClauseTuple

  %% Given a list of ActionClauseTuples, construct the summand that selects the
  %% right pc for the received command.
  constructClauseSelectors(ActionClauseTuple*, ProcName) -> ProcExpr

  %% Conversion functions for the statements inside action clauses.
  convertStatements(FSMStatement*, ProcName, PC, PC, PC)  -> <ProcExpr,PC>
  convertStatement(FSMStatement, ProcName, PC, PC, PC)    -> <ProcExpr,PC>

  %% Helper function for the translation of if statements.
  insertIfBlockingWaiter(ProcName, PC)                    -> ProcExpr

  %% Helpers for the generation of bottom monitors.
  inAnyState(MId*)                                                -> DataExpr
  createObedientCommandAcceptor(FSMActionClause*, ProcName, MId*)     -> ProcExpr

  %% For the when clauses we need to add a clause describing that we are in a certain
  %% state.
  isStateCheck(MId)      -> DataExpr
  isStateCheck(MId, MId) -> DataExpr
  isCommandCheck(MId)    -> DataExpr
  "isStateCheck"         -> DataExpr {reject}
  "isCommandCheck"       -> DataExpr {reject}


  %% Function to prepend 'is_' to an identifier ( MYID => is_MYID ).
  toMcrlIsFunction(MId) -> MId

  %% Convert a name of a state into an StateName as we will use in Mcrl2 ( OFF => S_OFF ).
  toMcrlStateName(MId) -> MId
  "toMcrlStateName"    -> DataExpr {reject}
  "toMcrlStateName"    -> MId {reject}

  %% Convert a name of a command into a CommandName as we wil use in Mcrl2 ( OFF => C_OFF ).
  toMcrlCmdName(MId)   -> MId
```

```
%%%%%%%%%%%% Sort generation

%% These functions are required to generate the sort declaratons from the FSM classes.

%% First functions to create a declaration that is used in a struct from the
%% name of the class/state/action. So from some class myClass it generates :
%% 'Class ? is_myClass'. Similarly for states and actions.
convertClassNamesToTypeConstrDecl(MId*)   -> {ConstrDecl "|"}+
convertStateNamesToStateConstrDecl(MId*)  -> {ConstrDecl "|"}+
convertActionNamesToCmdConstrDecl(MId*)   -> {ConstrDecl "|"}+

%% The following functions are traversal functions that simply gather all definitions
%% of a ClassName/StateName/Action name in the FSM classes that were supplied.
collectClasses(FSMSpecification,MId*)       -> MId* {traversal(accu,top-down,continue)}
collectClasses(FSMClass,MId*)               -> MId* {traversal(accu,top-down,continue)}
collectClasses(FSMChildrenAnySpecific,MId*) -> MId* {traversal(accu,top-down,continue)}
collectClasses(FSMChildrenAllSpecific,MId*) -> MId* {traversal(accu,top-down,continue)}

collectStates(FSMSpecification, MId*)     -> MId* {traversal(accu,top-down,continue)}
collectStates(FSMStateClause+, MId*)      -> MId* {traversal(accu,top-down,continue)}
collectStates(FSMStateClause, MId*)       -> MId* {traversal(accu,top-down,continue)}
collectStates(FSMWhenClause*, MId*)       -> MId* {traversal(accu,top-down,continue)}
collectStates(FSMWhenClause, MId*)        -> MId* {traversal(accu,top-down,continue)}
collectStates(FSMStateNameSpec, MId*)     -> MId* {traversal(accu,top-down,continue)}

collectCommands(FSMSpecification,MId*)    -> MId* {traversal(accu,top-down,continue)}
collectCommands(FSMActionClause,MId*)     -> MId* {traversal(accu,top-down,continue)}

collectActionNames(FSMSpecification, MId*) -> MId* {traversal(accu,top-down,continue)}
collectActionNames(FSMActionClause*, MId*) -> MId* {traversal(accu,top-down,continue)}

%% Function to add something to a set such that we do not introduce duplicates.
addToSet(MId,MId*)                        -> MId*

%%%%%%%%%%%% ProcNames generation
mcrl2PTypesFromProcSpecs(ProcSpec+) -> {ConstrDecl "|"}+
mcrl2PTypesFromProcDecls(ProcDecl+) -> {ConstrDecl "|"}+

variables
"$mid"[0-9]*                    -> MId
"$mid+"[0-9]*                   -> MId+
"$mid*"[0-9]*                   -> MId*
"$mids"[0-9]*                   -> MIds

"$fsmSpec"[0-9]*                -> FSMSpecification
"$fsmClass"[0-9]*               -> FSMClass
"$fsmClass+"[0-9]*              -> FSMClass+
"$fsmState"[0-9]*               -> FSMStateClause
"$fsmState+"[0-9]*              -> FSMStateClause+
"$fsmWhenClause"[0-9]*          -> FSMWhenClause
"$fsmWhenClause+"[0-9]*         -> FSMWhenClause+
"$fsmWhenClause*"[0-9]*         -> FSMWhenClause*
"$fsmActionClause"[0-9]*        -> FSMActionClause
"$fsmActionClause*"[0-9]*       -> FSMActionClause*
"$fsmExpr"[0-9]*                -> FSMExpression
"$fsmExpr*"[0-9]*               -> FSMExpression*
"$fsmStatement"[0-9]*           -> FSMStatement
"$fsmStatement+"[0-9]*          -> FSMStatement+
"$fsmStatement*"[0-9]*          -> FSMStatement*
"$fsmReferer"[0-9]*             -> FSMReferer

"$fsmClassName"[0-9]*           -> MId

"$fsmChildrenSpec"[0-9]*        -> FSMChildrenSpec
"$fsmStateNameSpec"[0-9]*       -> FSMStateNameSpec
"$fsmStateNameSpecs*"[0-9]*     -> {FSMStateName ","}*
"$fsmChildrenAnySpecific"[0-9]* -> FSMChildrenAnySpecific
"$fsmChildrenAllSpecific"[0-9]* -> FSMChildrenAllSpecific
"$fsmChildrenAnyFwChildren"[0-9]* -> FSMChildrenAnyFwChildren
"$fsmChildrenAllFwChildren"[0-9]* -> FSMChildrenAllFwChildren
"$fsmActionName"[0-9]*          -> MId %% must be MId for toMcrlCommandName

"$fsmStateName"[0-9]*           -> MId
"$fsmCurrentState"[0-9]*        -> MId
"$fsmNewState"[0-9]*            -> MId
"$fsmStateNames"[0-9]*          -> {FSMStateName ","}+
"$mcrl2CurrentState"[0-9]*      -> MId
"$mcrl2NewState"[0-9]*          -> MId

"$actionClauseTuple"[0-9]*      -> ActionClauseTuple
"$actionClauseTuple+"[0-9]*     -> ActionClauseTuple+
"$actionClauseTuple*"[0-9]*     -> ActionClauseTuple*
```

```
  "$procDecl"[0-9]*                -> ProcDecl
  "$procDecl+"[0-9]*               -> ProcDecl+
  "$procDecl*"[0-9]*               -> ProcDecl*
  "$dataExpr"[0-9]*                -> DataExpr
  "$dataExprs"[0-9]*               -> {DataExpr ","}+
  "$procExpr"[0-9]*                -> ProcExpr
  "$procSpec"[0-9]*                -> ProcSpec
  "$procSpec+"[0-9]*               -> ProcSpec+
  "$procSpec*"[0-9]*               -> ProcSpec*
  "$mcrl2Command"                  -> MId
  "$mcrlActionCondition"           -> DataExpr
  "$mcrlActionEffect"              -> ProcExpr

  "$procName"[0-9]*                -> MId

  "$pc"[0-9]*                      -> Integer
  "$start_pc"[0-9]*                -> Integer
  "$avail_pc"[0-9]*                -> Integer
  "$jump_pc"[0-9]*                 -> Integer
  "$then_pc"[0-9]*                 -> Integer
  "$else_pc"[0-9]*                 -> Integer
  "$end_pc"[0-9]*                  -> Integer

  "$idsDecls"                      -> {IdsDecl ","}+

  "$b"                             -> BoolCon

lexical variables
  "#midHead"[0-9]*                 -> [a-zA-Z\_]
  "#midTailChar"[0-9]*             -> ([a-zA-Z0-9\_\'])
  "#midTail"[0-9]*                 -> ([a-zA-Z0-9\_\'])*
```

A.6 cfsm2mcrl2.asf

```
equations

[cfsm2mcrl2-1]
cfsm2mcrl2($fsmClass+) = fsmClasses2Mcrl2Procs($fsmClass+, false)

  %% Convertor function to convert a FSM class to an MCRL2 specfication, with the added
  %% property that the result will be a bottom monitor. That is, it has no children and
  %% whenever it would normally check the state of children, it will instead check
  %% randomStateChanges.
[cfsm2mcrl2bm-1]
cfsm2mcrl2bm($fsmClass+) = fsmClasses2Mcrl2Procs($fsmClass+, true)

%% Convert a number of FSM classes into processes.
[convertSpec-single]
fsmClasses2Mcrl2Procs($fsmClass, $b) = fsmClass2Mcrl2Proc($fsmClass, $b)

[convertSpec-multi]
fsmClasses2Mcrl2Procs($fsmClass $fsmClass+, $b) =
    fsmClass2Mcrl2Proc($fsmClass, $b) fsmClasses2Mcrl2Procs($fsmClass+, $b)

%% In our main function we define the process instance and process declaration.

[fsmClassName2ProcName-bm]
fsmClassName2ProcName($mid, true) = $mid %%concat($mid, _BM)

[fsmClassName2ProcName-nobm]
fsmClassName2ProcName($mid, false) = $mid


[fsmClass2Mcrl2Proc-1]
$procName := fsmClassName2ProcName($fsmClassName, $b),
$procExpr := convertStates($fsmState+, $procName, $b, collectStates($fsmState+, )),
$procSpec := proc $procName(self: Id, parent: Id, s: State, chs: Children, phase: Phase, aArgs: ActPhaseArgs) =
    $procExpr +
    insertGenericClauses($procName);
===>
fsmClass2Mcrl2Proc(class: $FWPART_$TOP$ $fsmClassName $fsmState+, $b) = $procSpec


%% Function for converting a list of states. Each state translation translates to a Process Expression, so
%% this means a list should be translated using the alternative ( + ) operator
[convertStates-1-element]
convertStates($fsmState, $procName, $b, $mid*) = convertState($fsmState, $procName, $b, $mid*)

[convertStates-list]
convertStates($fsmState $fsmState+, $procName, $b, $mid*) =
    convertState($fsmState, $procName, $b, $mid*) +
    convertStates($fsmState+, $procName, $b, $mid*)
```

```
%% Functon to convert a single state.
[convertState-nobm]
<$actionClauseTuple*, $pc> := gatherComponentsFromActionClauses($fsmActionClause*, $procName, $fsmCurrentState, 1)
===>
convertState(state: $fsmCurrentState $fsmWhenClause* $fsmActionClause*, $procName, false, $mid*) =
    (
    % ===========
    % BEGIN STATE

    % ------------------
    % BEGIN WHEN CLAUSES

    convertWhenClauses($fsmWhenClause*, $procName, $fsmCurrentState, $actionClauseTuple*) +

    % END WHEN CLAUSES
    % ----------------

    % --------------------
    % BEGIN ACTION CLAUSES

    % These are the rules:
    % pc(aArgs) == 0                        => no command received yet
    % pc(aArgs) > 0                         => command received, executing action clause
    % pc(aArgs) == -1 && cq(aArgs) != []    => action clause executed, but still must send commands
    % pc(aArgs) == -1 && cq(aArgs) == []    => action clause executed

    %% Since the FSM language allows for an arbitrary amount of statements and
    %% an arbitrary amount of (nested) if-statements, we cannot simply do a
    %% sequential translation. It is for this reason that we use a label to
    %% identify the translation of every statement. After executing a
    %% statement, a program counter is set to the label of the statement which
    %% should be executed next. There are two special cases here:
    %%   * Label 0, the clause selector. In the action phase, we always first
    %%     have pc == 0. When we receive a command, the clause selector
    %%     determines the label of the first statement of the action clause
    %%     that should handle the command. The program count is then set to
    %%     this label.
    %%   * Label -1, end of action. After executing an action, the program
    %%     counter is set to -1 to signify that we should now empty the
    %%     sendqueue and move to the when phase.

    %% Examples can be found in the translation function of the if-statement.

    % BEGIN INITIALIZATION CHECK

    ((isStateCheck($fsmCurrentState)) && (isActPhase(phase)) && (!(initialized(chs))) &&
     (pc(aArgs) == 0) && (nrf(aArgs) == [])) ->
        start_initialization(self).
        $procName(self, parent, s, chs, phase,
                  actArgs([], children_to_ids(chs), 0, rsc(aArgs))) <>

    % END INITIALIZATION CHECK

    % BEGIN CLAUSE SELECTOR

    ((initialized(chs)) ->
    (
    (((isStateCheck($fsmCurrentState)) && (isActPhase(phase)) && (cq(aArgs) == []) && (pc(aArgs) == 0)) ->
        sum c:Command.(
            rc(parent, self, c).
            constructClauseSelectors($actionClauseTuple*, $procName)
    )) +
    % END CLAUSE SELECTOR

    combineActionClauseComponents($actionClauseTuple*, $procName, $fsmCurrentState)
    ))

    % END ACTION CLAUSES
    % ------------------

    % END STATE
    % =========
    )

%% Functon to convert a single state (bottom monitor variant).
[convertState-bm]
$dataExpr := inAnyState(collectStates($fsmWhenClause*, )),
$procExpr := createObedientCommandAcceptor($fsmActionClause*, $procName, $mid*)
===>
convertState(state: $fsmCurrentState $fsmWhenClause* $fsmActionClause*, $procName, true, $mid*) =
    (
    % ===========
    % BEGIN STATE

    % --------------------
```

```
    % BEGIN ACTION CLAUSES

    ((isStateCheck($fsmCurrentState)) ->
    sum c: Command.
    (
    rc(parent, self, c).
    $procExpr
    )
    )

    % END ACTION CLAUSES
    % ------------------

    % END STATE
    % =========
    )


%%%%%%%%%%%%%%%%%%% When Clauses Stuff

[convertWhenClauses-empty]
$mcrl2CurrentState := toMcrlStateName($fsmCurrentState)
===>
convertWhenClauses(, $procName, $fsmCurrentState, $actionClauseTuple*) =
    (
    % BEGIN WHEN FALLTHROUGH

    (((isStateCheck($fsmCurrentState)) && (isWhenPhase(phase))) ->
    ss(self, parent, s).
    move_phase(self, ActionPhase).
    $procName(self, parent, s, chs, ActionPhase, reset(aArgs)))

    % END WHEN FALLTHROUGH
    )

[convertReferer-moveto]
$mcrl2NewState := toMcrlStateName($fsmNewState)
===>
convertReferer(move_to $fsmNewState, $procName, $mcrl2CurrentState, $actionClauseTuple*) =
    move_state(self, $mcrl2NewState).
    $procName(self, parent, $mcrl2NewState, chs, phase, aArgs)

[convertReferer-do]
<$fsmActionName, $start_pc, $mcrlActionCondition, $mcrlActionEffect> := getActionClauseTupleForActionName($actionClauseTuple*, $fsmActionName)
===>
convertReferer(do $fsmActionName, $procName, $mcrl2CurrentState, $actionClauseTuple*) =
    move_phase(self, ActionPhase).
    $mcrlActionEffect

%% Note: the empty list is not allowed here since there must always be an corresponding action. If not, the FSM is inconsistent.
[getActionClauseTupleForActionName-many-match]
<$fsmActionName, $start_pc, $mcrlActionCondition, $mcrlActionEffect> := $actionClauseTuple
===>
getActionClauseTupleForActionName($actionClauseTuple $actionClauseTuple*, $fsmActionName) =
    $actionClauseTuple

[getActionClauseTupleForActionName-many-nomatch]
<$fsmActionName2, $start_pc, $mcrlActionCondition, $mcrlActionEffect> := $actionClauseTuple,
$fsmActionName1 != $fsmActionName2
===>
getActionClauseTupleForActionName($actionClauseTuple $actionClauseTuple*, $fsmActionName1) =
    getActionClauseTupleForActionName($actionClauseTuple*, $fsmActionName1)

%% When we have multiple elements in our list of when clauses we translate into
%% a form of 'c -> a.X <> b' in which 'b' is the translation of the remaining
%% when clauses
[convertWhenClauses-many]
$mcrl2CurrentState := toMcrlStateName($fsmCurrentState)
===>
convertWhenClauses(when ($fsmExpr) $fsmReferer $fsmWhenClause*, $procName, $fsmCurrentState, $actionClauseTuple*) =
    (
    % BEGIN WHEN
    ((isStateCheck($fsmCurrentState)) && (isWhenPhase(phase)) &&
     (convertExpr($fsmExpr))) ->
    convertReferer($fsmReferer, $procName, $mcrl2CurrentState, $actionClauseTuple*) <>

    % END WHEN
    (convertWhenClauses($fsmWhenClause*, $procName, $fsmCurrentState, $actionClauseTuple*))
    )


%%%%%%%%%%%%%%%%%%%%%% Action Clauses Stuff

[gatherComponentsFromActionClauses-empty]
gatherComponentsFromActionClauses(, $procName, $fsmCurrentState, $pc) =
```

```
    <, $pc>

[gatherComponentsFromActionClauses-many]
$start_pc1 := $avail_pc1,
$avail_pc2 := $avail_pc1 + 1,
$mcrl2Command := toMcrlCmdName($fsmActionName),
<$procExpr, $avail_pc3> := convertStatements($fsmStatement*, $procName, $start_pc1, -1, $avail_pc2),
<$actionClauseTuple*, $avail_pc4> :=
    gatherComponentsFromActionClauses($fsmActionClause*, $procName, $fsmCurrentState, $avail_pc3)
===>
gatherComponentsFromActionClauses(action: $fsmActionName $fsmStatement* $fsmActionClause*,
                                  $procName, $fsmCurrentState, $avail_pc1) =
    <
    <
    $fsmActionName,

    $start_pc1,

    ((isStateCheck($fsmCurrentState)) && (isActPhase(phase)) && (cq(aArgs) == [])),

    ($procExpr)
    >
    $actionClauseTuple*, $avail_pc4>

[combineActionClauseComponents-empty]
combineActionClauseComponents(, $procName, $fsmCurrentState) = delta

[combineActionClauseComponents-many]
<$fsmActionName, $start_pc, $mcrlActionCondition, $mcrlActionEffect> := $actionClauseTuple
===>
combineActionClauseComponents($actionClauseTuple $actionClauseTuple*, $procName, $fsmCurrentState) =
    (
    % BEGIN ACTION

    ($mcrlActionCondition ->
    $mcrlActionEffect) +

    % END ACTION

    (combineActionClauseComponents($actionClauseTuple*, $procName, $fsmCurrentState)))


[constructClauseSelectors-empty]
constructClauseSelectors(, $procName) =
    % BEGIN ACTION FALLTHROUGH

    ss(self, parent, s).
    ignored_command(self, c).
    $procName(self, parent, s, chs, phase, update_pc(aArgs, -1))

    % END ACTION FALLTHROUGH


[constructClauseSelectors-many]
constructClauseSelectors(<$fsmActionName, $start_pc, $mcrlActionCondition, $mcrlActionEffect> $actionClauseTuple*,
                         $procName) =
    (isCommandCheck($fsmActionName) -> $procName(self, parent, s, chs, phase, update_pc(aArgs, $start_pc)) <> (
           constructClauseSelectors($actionClauseTuple*, $procName)))


[createObedientCommandAcceptor-empty]
createObedientCommandAcceptor(, $procName, $mid*) = constructClauseSelectors(, $procName)

[createObedientCommandAcceptor-many-nomatch]
contains($fsmActionName, $mid*) == false
===>
createObedientCommandAcceptor(action: $fsmActionName $fsmStatement* $fsmActionClause*, $procName, $mid*) =
    createObedientCommandAcceptor($fsmActionClause*, $procName, $mid*)

[createObedientCommandAcceptor-many-match]
contains($fsmActionName, $mid*) == true,
$mcrl2NewState := toMcrlStateName($fsmActionName),
$mid := toMcrlCmdName($fsmActionName)
===>
createObedientCommandAcceptor(action: $fsmActionName $fsmStatement* $fsmActionClause*, $procName, $mid*) =
    ((c == $mid) ->
    ss(self, parent, $mcrl2NewState).
    move_state(self, $mcrl2NewState).
    $procName(self, parent, $mcrl2NewState, chs, WhenPhase, (reset(aArgs)))
    <>
    createObedientCommandAcceptor($fsmActionClause*, $procName, $mid*))

%%%%%%%%%%%%%%%% Statements

[convertStatements-empty]
```

```
%% Some FSMs in the bottommost layer might have actions which do not contain any statements.
convertStatements(, $procName, $start_pc, $jump_pc, $avail_pc) =
    <
    (
    % BEGIN STATEMENT NOOP
    ((pc(aArgs) == $start_pc) ->
         noop_statement(self).
         ($procName(self, parent, s, chs, phase, update_pc(aArgs, $jump_pc))))
    % END STATEMENT NOOP
    )
    , $avail_pc>

[convertStatements-single]
%% The final statement in a block should jump to the next block (indicated by $jump_pc).
convertStatements($fsmStatement, $procName, $start_pc, $jump_pc, $avail_pc) =
    convertStatement($fsmStatement, $procName, $start_pc, $jump_pc, $avail_pc)

[convertStatements-multiple]
$start_pc2 := $avail_pc1,
$avail_pc2 := $avail_pc1 + 1,
<$procExpr1, $avail_pc3> := convertStatement($fsmStatement, $procName, $start_pc1, $start_pc2, $avail_pc2),
<$procExpr2, $avail_pc4> := convertStatements($fsmStatement+, $procName, $start_pc2, $jump_pc, $avail_pc3)
===>
convertStatements($fsmStatement $fsmStatement+, $procName, $start_pc1, $jump_pc, $avail_pc1) =
    <$procExpr1 +
     $procExpr2, $avail_pc4>

[convertStatement-do]
convertStatement(do $fsmActionName $fsmChildrenSpec, $procName, $start_pc, $jump_pc, $avail_pc) =
    <
    (
    % BEGIN STATEMENT DO
    ((pc(aArgs) == $start_pc) ->
         queue_messages(self).
         ($procName(self, parent, s, chs, phase,
                    actArgs(send_command(toMcrlCmdName($fsmActionName),
                                         convertChildrenSpec($fsmChildrenSpec)), [],  $jump_pc, rsc(aArgs)))))
    % END STATEMENT DO
    )
    , $avail_pc>

[convertStatement-moveto]
$mcrl2NewState := toMcrlStateName($fsmNewState)
===>
convertStatement(move_to $fsmNewState, $procName, $start_pc, $jump_pc, $avail_pc) =
    <
    (
    % BEGIN STATEMENT MOVE_TO
    ((pc(aArgs) == $start_pc) ->
          (ss(self, parent, $mcrl2NewState).
           move_phase(self, WhenPhase).
           $procName(self, parent, $mcrl2NewState, chs, ActionPhase, reset(aArgs))))
    % END STATEMENT MOVE_TO
    )
    , $avail_pc>

[insertIfBlockingWaiter-1]
insertIfBlockingWaiter($procName, $pc) =
sum s1:State.(
         rs(id(head(busy_children(chs))), self, s1).
         $procName(self, parent, s,
                   update_busy(id(head(busy_children(chs))),
                               false,
                               update_state(id(head(busy_children(chs))), s1, chs)),
                   phase, update_pc(aArgs, $pc)))

[convertStatement-ifthenend]
$start_pc2 := $avail_pc1,
$avail_pc2 := $avail_pc1 + 1,
<$procExpr1, $avail_pc3> :=
    convertStatements($fsmStatement+, $procName, $start_pc2, $jump_pc, $avail_pc2),
$procExpr2 :=
    (
    % BEGIN STATEMENT IF-THEN-ENDIF
    ((pc(aArgs) == $start_pc1) ->
    (
    (busy_children(chs) != []) ->
      (
      insertIfBlockingWaiter($procName, $start_pc1)
      )
    <>
      (
      ((convertExpr($fsmExpr)) ->
        enter_then_clause(self).
        $procName(self, parent, s, chs, phase, update_pc(aArgs, $start_pc2)) <>
```

```
        skip_then_clause(self).
        $procName(self, parent, s, chs, phase, update_pc(aArgs, $jump_pc)))
      )
      )) +

      (
      % BEGIN THEN
       $procExpr1
      % END THEN
      )

    % END STATEMENT IF-THEN-ENDIF
    )
===>
convertStatement(if ( $fsmExpr ) then $fsmStatement+ endif, $procName, $start_pc1, $jump_pc, $avail_pc1) =
    <
    $procExpr2,
    $avail_pc3
    >

[convertStatement-ifthenelseend]
%% Suppose we have the following FSM statements (pseudocode):
%%
%%    do STANDBY c1
%%    if b then
%%        do ON c1
%%        do ON c2
%%        do STANDBY c3
%%    else
%%        do OFF c1
%%        move_to ERROR
%%        do OFF c2
%%    endif
%%    do ON c4
%%
%% We will now give a simplified translation of these statements. For
%% every statement, we give the label of the statement at the beginning
%% of the line, and the label of the next statement at the end of the line,
%% along with some explanations.
%%
%% We assume that:
%%  * start_pc1 = 5
%%  * jump_pc   = 6
%%  * avail_pc  = 10
%%
%% The simplified translation follows:
%%
%%  5.  queue STANDBY to c1                    (-> 10, since 10 is the first available label)
%%  10. IF   there is a busy child
%%      THEN get the new state of a busy child  (-> 10, i.e. loop until there are no busy children)
%%      ELSE IF b
%%           THEN enter_then_clause(self)       (-> 12; note that 11 is reserved for the statement after the if statement)
%%           ELSE enter_else_clause(self)       (-> 13)
%%  12. queue ON command to c1                  (-> 14; note that 13 is taken by the first statement of the else clause)
%%  14. queue ON command to c2                  (-> 15)
%%  15. queue STANDBY command to c3             (-> 11; end of this block, so jump to the statement after the if statement)
%%  13. queue OFF command to c1                 (-> 16, since 14-15 are used by the then-block)
%%  16. move to the ERROR state                 (-> -1; special case: after a move_to we leave the action phase)
%%  17. queue OFF command to c2                 (-> 11; unreachable due to the move_to on the previous line)
%%  11. send ON command to c4                   (->  6; last statement in the list, so we jump to jump_pc)
%%
$start_pc2 := $avail_pc1,
$start_pc3 := $avail_pc1 + 1,
$avail_pc2 := $avail_pc1 + 2,
<$procExpr1, $avail_pc3> :=
    convertStatements($fsmStatement+1, $procName, $start_pc2, $jump_pc, $avail_pc2),
<$procExpr2, $avail_pc4> :=
    convertStatements($fsmStatement+2, $procName, $start_pc3, $jump_pc, $avail_pc3),
$procExpr3 :=
    (
    % BEGIN STATEMENT IF-THEN-ELSE-ENDIF
    ((pc(aArgs) == $start_pc1) ->
    (
    (busy_children(chs) != []) ->
      (
      insertIfBlockingWaiter($procName, $start_pc1)
      )
    <>
      (
      ((convertExpr($fsmExpr)) ->
        enter_then_clause(self).
        $procName(self, parent, s, chs, phase, update_pc(aArgs, $start_pc2)) <>
        enter_else_clause(self).
        $procName(self, parent, s, chs, phase, update_pc(aArgs, $start_pc3)))
      )
```

```
      )) +

      (
      % BEGIN THEN
       $procExpr1
      % END THEN
      ) +

      (
      % BEGIN ELSE
      $procExpr2
      % END ELSE
      )

    % END STATEMENT IF-THEN-ELSE-ENDIF
    )
===>
convertStatement(if ( $fsmExpr ) then $fsmStatement+1 else $fsmStatement+2 endif, $procName, $start_pc1, $jump_pc, $avail_pc1) =
    <
    $procExpr3,
    $avail_pc4
    >

%%%%%%%%%%%%%% Expressions

%% Conversion of expressions. We have two compound types: and-expressions and
%% or-expressions:
[convertExpr-and]
convertExpr($fsmExpr0 and $fsmExpr1) = convertExpr($fsmExpr0) && convertExpr($fsmExpr1)
[convertExpr-or]
convertExpr($fsmExpr0 or $fsmExpr1) = convertExpr($fsmExpr0) || convertExpr($fsmExpr1)

%% And the expressions which check if certain children are in some specific
%% state. These depend on the specified children (either all children, any
%% child, all children of a certain type, or any child of a certain type).

%% For the all children/any child we simply use the all_in_state(chs,stateType)
%% translation.
[convertExpr-allchildren]
convertExpr($fsmChildrenAllFwChildren in_state $fsmStateNameSpec) =
    all_in_state(convertChildrenSpec($fsmChildrenAllFwChildren), convertStateNameSpec($fsmStateNameSpec))
[convertExpr-anychildren]
convertExpr($fsmChildrenAnyFwChildren in_state $fsmStateNameSpec) =
    any_in_state(convertChildrenSpec($fsmChildrenAnyFwChildren), convertStateNameSpec($fsmStateNameSpec))

%% For the translation of 'all children of type T' we should make sure to use a
%% subset of our children 'chs'. To do that the convertChildrenSpec function
%% will make sure we apply the 'filter_children' method.
[convertExpr-allinstatespecific]
convertExpr($fsmChildrenAllSpecific in_state $fsmStateNameSpec) =
    all_in_state(convertChildrenSpec($fsmChildrenAllSpecific), convertStateNameSpec($fsmStateNameSpec))
[convertExpr-anyinstatespecific]
convertExpr($fsmChildrenAnySpecific in_state $fsmStateNameSpec) =
    any_in_state(convertChildrenSpec($fsmChildrenAnySpecific), convertStateNameSpec($fsmStateNameSpec))

[convertExpr-notallchildren]
convertExpr(not ($fsmChildrenAllFwChildren) in_state $fsmStateNameSpec) =
    !(all_in_state(convertChildrenSpec($fsmChildrenAllFwChildren), convertStateNameSpec($fsmStateNameSpec)))

[convertExpr-notallinstatespecific]
convertExpr(not ($fsmChildrenAllSpecific) in_state $fsmStateNameSpec) =
    !(all_in_state(convertChildrenSpec($fsmChildrenAllSpecific), convertStateNameSpec($fsmStateNameSpec)))


%% We now repeat these translations for the not_in_state expressions.
[convertExpr-allchildren-not]
convertExpr($fsmChildrenAllFwChildren not_in_state $fsmStateNameSpec) =
    !(any_in_state(convertChildrenSpec($fsmChildrenAllFwChildren), convertStateNameSpec($fsmStateNameSpec)))
[convertExpr-anychildren-not]
convertExpr($fsmChildrenAnyFwChildren not_in_state $fsmStateNameSpec) =
    !(all_in_state(convertChildrenSpec($fsmChildrenAnyFwChildren), convertStateNameSpec($fsmStateNameSpec)))
[convertExpr-allinstatespecific-not]
convertExpr($fsmChildrenAllSpecific not_in_state $fsmStateNameSpec) =
    !(any_in_state(convertChildrenSpec($fsmChildrenAllSpecific), convertStateNameSpec($fsmStateNameSpec)))
[convertExpr-anyinstatespecific-not]
convertExpr($fsmChildrenAnySpecific not_in_state $fsmStateNameSpec) =
    !(all_in_state(convertChildrenSpec($fsmChildrenAnySpecific), convertStateNameSpec($fsmStateNameSpec)))


%% Expressions can have brackets, simply leave them as they are and translate
%% the epxression inside them.
[convertExpr-bracket]
convertExpr(($fsmExpr)) = (convertExpr($fsmExpr))

%% Apply the filter on the childrenlist.
```

```
[convertChildrenSpec-alltype]
convertChildrenSpec($ALL$ $fsmClassName) = filter_children(chs, concat($fsmClassName, _CLASS))
[convertChildrenSpec-anytype]
convertChildrenSpec($ANY$ $fsmClassName) = filter_children(chs, concat($fsmClassName, _CLASS))
[convertChildren-all]
convertChildrenSpec($fsmChildrenAllFwChildren) = chs
[convertChildren-any]
convertChildrenSpec($fsmChildrenAnyFwChildren) = chs

%% Conversion of the StateNameSpec in the FSMs into a list of states. A
%% stateNameSpec is either a simple state name:
[convertStateNameSpec-single]
convertStateNameSpec($fsmStateName) = [toMcrlStateName($fsmStateName)]
%% Or a set of state names in the form "{ name1, name2, ... }". We can then
%% distinguish two cases: We have exactly one statename or multiple statenames
[convertStateNameSpec-single-in-multiple]
convertStateNameSpec({$fsmStateName}) = [toMcrlStateName($fsmStateName)]
[convertStateNameSpec-multiple]
[ $dataExprs ] := convertStateNameSpec({$fsmStateNames})
===>
convertStateNameSpec({$fsmStateName, $fsmStateNames}) = [toMcrlStateName($fsmStateName), $dataExprs]

[inAnyState-empty]
inAnyState() = false
[inAnyState-one]
inAnyState($mid) = isStateCheck($mid, s1)
[inAnyState-many]
inAnyState($mid $mid+) = isStateCheck($mid, s1) || inAnyState($mid+)


%%%%%%%% Lexical Stuff %%%%%%%%%%

%% Create a check if the currentState is state id, i.e. convert idStateCheck(myState) into: isMyState(s).
[isStateCheck-2]
mid(#midHead #midTail) := toMcrlIsFunction(toMcrlStateName($mid1))
===>
isStateCheck($mid1, $mid2) = mid(#midHead #midTail)($mid2)

[isStateCheck-1]
isStateCheck($mid) = isStateCheck($mid, s)

%% Same for command checks.
[isCommand-c]
mid(#midHead #midTail) := toMcrlIsFunction(toMcrlCmdName($mid))
===>
isCommandCheck($mid) = mid(#midHead #midTail)(c)

%% Function to prepend the is to the function name.
[toMcrlIsFunc]
toMcrlIsFunction(mid(#midHead #midTail)) = mid(is #midHead #midTail)

[toMcrlState-1]
toMcrlStateName(mid(#midHead #midTail)) = concat(S_, mid(#midHead #midTail))

[toMcrlCmd-1]
toMcrlCmdName(mid(#midHead #midTail)) = concat(C_, mid(#midHead #midTail))

%%%%%%%% Create Sort declaration stuff

%% We generate the sort declarations for the Process Type, State and Command.
%% All these sorts are structs. So for each one collect all names of Classes
%% (=Process Types)/States/Actions and create the struct declarations from
%% these names

[generateSorts]
fsmGenerateSorts($fsmClass+) =
    PType = struct convertClassNamesToTypeConstrDecl(collectClasses($fsmClass+,));
    State = struct S_FSM_UNINITIALIZED ? isS_FSM_UNINITIALIZED | convertStateNamesToStateConstrDecl(collectStates($fsmClass+,));
    Command = struct convertActionNamesToCmdConstrDecl(collectCommands($fsmClass+,));

%% Convert the state names from the form 'statename' into 'S_statename ? isS_statename'
[convertStateNamesToSortDecl-single]
convertStateNamesToStateConstrDecl($mid) = toMcrlStateName($mid) ? toMcrlIsFunction(toMcrlStateName($mid))
[convertStateNamesToSortDecl-multi]
convertStateNamesToStateConstrDecl($mid $mid+) = convertStateNamesToStateConstrDecl($mid) | convertStateNamesToStateConstrDecl($mid+)

%% convert actions/commands from 'commandname' into 'C_commandname ? isC_commandname'
[convertActionNamesToSortDecl-single]
convertActionNamesToCmdConstrDecl($mid) = toMcrlCmdName($mid) ? toMcrlIsFunction(toMcrlCmdName($mid))
[convertActionNamesToSortDecl-multi]
convertActionNamesToCmdConstrDecl($mid $mid+) = convertActionNamesToCmdConstrDecl($mid) | convertActionNamesToCmdConstrDecl($mid+)

%% convert the class names from 'classname' into 'classname ? isclassname'
[convertClassNamesToSortDecl-single]
convertClassNamesToTypeConstrDecl($mid) = $mid ? toMcrlIsFunction($mid)
[convertClassNamesToTypeSortDecl-multi]
```

```
convertClassNamesToTypeConstrDecl($mid $mid+) = convertClassNamesToTypeConstrDecl($mid) | convertClassNamesToTypeConstrDecl($mid+)

%% Traversal functions to collect the classnames, action-names and statenames.

[collect-class-definition]
collectClasses(class: $FWPART_$TOP$ $fsmClassName $fsmState+, $mid*) = addToSet($fsmClassName, $mid*)

[collect-class-exprall]
collectClasses($ALL$$fsmClassName, $mid*) = addToSet(concat($fsmClassName, _CLASS), $mid*)

[collect-class-exprany]
collectClasses($ANY$$fsmClassName, $mid*) = addToSet(concat($fsmClassName, _CLASS), $mid*)

[collect-command]
collectCommands(action: $fsmActionName $fsmStatement+, $mid*) = addToSet($fsmActionName, $mid*)

[collect-state]
collectStates(state: $fsmStateName $fsmWhenClause+ $fsmActionClause*, $mid*) = addToSet($fsmStateName, $mid*)

[collect-state-when]
collectStates(when ( $fsmExpr ) move_to $fsmStateName, $mid*) = addToSet($fsmStateName, $mid*)

[collect-state-statenamespec]
$mid*1 := collectStates({ $fsmStateNameSpecs* }, $mid*)
===>
collectStates( { $fsmStateName, $fsmStateNameSpecs*}, $mid*) = addToSet($fsmStateName, $mid*1)

[collect-state-statenamespec-1-element]
collectStates( $fsmStateName, $mid*) = addToSet($fsmStateName, $mid*)


%%%%%%%% Constructing a PType for an mcrl2 specification.
[mcrl2GetPTypes-1]
mcrl2GetPTypes($procSpec+) =
    PType = struct mcrl2PTypesFromProcSpecs($procSpec+);

[mcrl2PTypesFromProcSpecs-one]
mcrl2PTypesFromProcSpecs(proc $procDecl+) =
    mcrl2PTypesFromProcDecls($procDecl+)

[mcrl2PTypesFromProcSpecs-many]
mcrl2PTypesFromProcSpecs(proc $procDecl+ $procSpec+) =
    mcrl2PTypesFromProcDecls($procDecl+) | mcrl2PTypesFromProcSpecs($procSpec+)

[mcrl2PTypesFromProcDecls-one-1]
mcrl2PTypesFromProcDecls($mid = $procExpr;) =
    convertClassNamesToTypeConstrDecl($mid)

[mcrl2PTypesFromProcDecls-many-1]
mcrl2PTypesFromProcDecls($mid = $procExpr; $procDecl+) =
    convertClassNamesToTypeConstrDecl($mid) | mcrl2PTypesFromProcDecls($procDecl+)

[mcrl2PTypesFromProcDecls-one-2]
mcrl2PTypesFromProcDecls($mid ( $idsDecls ) = $procExpr;) =
    convertClassNamesToTypeConstrDecl($mid)

[mcrl2PTypesFromProcDecls-many-2]
mcrl2PTypesFromProcDecls($mid ( $idsDecls ) = $procExpr; $procDecl+) =
    convertClassNamesToTypeConstrDecl($mid) | mcrl2PTypesFromProcDecls($procDecl+)

[addToSet-empty]
addToSet($mid,) = $mid
[addToSet-multisame]
addToSet($mid,$mid $mid*) = $mid $mid*
[addToSet-multidiff]
$mid != $mid1
===>
addToSet($mid,$mid1 $mid*) = $mid1 addToSet($mid, $mid*)
```

A.7 genericclauses.sdf

```
module genericclauses

imports basic/Comments
imports mcrlt

exports

context-free syntax

  %% Insert the generic code that sends the commands to the children.
  insertGenericClauses(MId) -> ProcExpr
```

```
hiddens
variables
  "$fsmClassName" -> MId
```

A.8 genericclauses.asf

```
equations

[insertGeneric]
insertGenericClauses($fsmClassName) =
    (
    % BEGIN GENERIC CLAUSES (shared by all states)
    % Whenever we are not sending a command to the children, a child may
    % spontaneously change its state due to a hardware event and send its
    % state upward. Such state-change messages are called notifications.

    % Notifications can occur in the following cases:
    %  (1) After initialization, while in the action phase:
    %  (1.a) We have not received a command yet in this action phase.
    %  (1.b) We are executing an action, or we finished executing an action but still have to send
    %        some commands.
    %  (2) During initialization.

    % Note that this implies that we never receive notifications during the
    % execution of the when phase (i) and we never receive notifcations
    % directly after we finish sending the last command after executing an
    % action (ii).

    % The rationale behind this is as follows:
    %  (i)  The execution of the when clauses is a noninteractive process: the
    %       system decides what the new state is, based only on *local*
    %       information.
    %  (ii) After sending the last command, the model moves immediately into
    %       the when phase. We should therefore not accept notifications at
    %       this point.

    % (1.a) We have initialized and we have not yet received a command in this
    %       action phase. We now accept notifications.
    sum id:Id.(sum s1:State.(((isActPhase(phase)) && (is_child(id, chs)) &&
                             (pc(aArgs) == 0) && (initialized(chs))) ->
       rs(id, self, s1).
       move_phase(self, WhenPhase).
       $fsmClassName(self, parent, s, update_busy(id, false, update_state(id, s1, chs)), WhenPhase, reset(aArgs))))
    +

    % (1.b) We are in the middle of executing an action, or we finished
    %       executing and still have to send some commands. We accept
    %       notifications, but we don't move to the when phase, since we still
    %       must execute one or more statements.
    sum id:Id.(sum s1:State.(((isActPhase(phase)) && (is_child(id, chs)) &&
           ((pc(aArgs) > 0) ||
            ((pc(aArgs) == -1) && (cq(aArgs) != [])))) ->
       rs(id, self, s1).
       $fsmClassName(self, parent, s,
                     update_busy(id,
                                 false,
                                 update_state(id, s1, chs)),
                     phase, aArgs))) +


    % Clause to send commands added by actions in the initialization phase.
    % Note that we keep track of the children which have not yet responded in
    % the nrf list.
    ((isActPhase(phase)) && (cq(aArgs) != []) && (!(initialized(chs)))) ->
        sc(self, id(head(cq(aArgs))), command(head(cq(aArgs)))).
            $fsmClassName(self, parent, s,
                          update_busy(id(head(cq(aArgs))), true, chs),
                          phase,
                          actArgs(tail(cq(aArgs)),
                                  (id(head(cq(aArgs)))|>(nrf(aArgs)), pc(aArgs), rsc(aArgs))) +


    % Clause to send commands added by actions after the initialization phase.
    % Note that we don't keep track of the children which have not yet
    % responded. Recepients are only marked busy and not added to the nrf list.
    ((isActPhase(phase)) && (cq(aArgs) != []) && initialized(chs)) ->
        sc(self, id(head(cq(aArgs))), command(head(cq(aArgs)))).
            $fsmClassName(self, parent, s,
                          update_busy(id(head(cq(aArgs))), true, chs),
                          phase,
                          actArgs(tail(cq(aArgs)), [], pc(aArgs), rsc(aArgs))) +
```

```
    % (2) Clause to receive the new states from the children during
    %     initialization.
80
    % Note that some children may spontaneously change state and send a
    % notification. This may cause us to receive more than one message from a
    % child while we wait for all children to respond. If a child sends two
    % state messages, we will only consider the last state when we process the
85  % when clauses.
    sum id:Id.(sum s1:State.(
            ((isActPhase(phase)) && (cq(aArgs) == []) && (nrf(aArgs) != []) && (is_child(id, chs)) &&
             (!(initialized(chs)))) ->
                   rs(id, self, s1).
90                 ((initialized(update_state(id,s1,chs))) ->
                        end_initialization(self).
                        $fsmClassName(self, parent, s, update_state(id,s1,chs), phase,
                                      actArgs(cq(aArgs), remove(id, nrf(aArgs)), -1, rsc(aArgs)
                                      )) <>
95                      $fsmClassName(self, parent, s, update_state(id,s1,chs), phase,
                                      actArgs(cq(aArgs), remove(id, nrf(aArgs)), -1, rsc(aArgs)
                                      ))))) +

    % Go to the when phase whenever all children are initialized, we executed
100 % an action and there are no pending messages.
    ((isActPhase(phase)) && (cq(aArgs) == []) && (initialized(chs)) && (pc(aArgs) == -1)) ->
        move_phase(self, WhenPhase).
        $fsmClassName(self, parent, s, chs, WhenPhase, reset(aArgs))

105 % END GENERIC CLAUSES
    )
```

B Wheel subsystem

```
1   class: $FWPART_$TOP$RPC_Wheel_CLASS
    !panel: CMS_RPCfwSupervisor/CMS_RPCfwSupervisorRPC_Wheel.pnl
        state: OFF  !color: FwStateOKNotPhysics
            when ( $ANY$FwCHILDREN in_state ERROR )  move_to ERROR
5
            when ( $ANY$FwCHILDREN in_state RAMPING )  move_to RAMPING
            when ( $ALL$FwCHILDREN in_state STANDBY )  move_to STANDBY

            when ( $ALL$FwCHILDREN in_state ON )  move_to ON
10
            when ( ( $ALL$FwCHILDREN not_in_state OFF ) and
           ( $ANY$FwCHILDREN in_state STANDBY ) )  move_to STANDBY
            action: STANDBY !visible: 1
                do STANDBY $ALL$FwCHILDREN
15          action: OFF !visible: 1
                do OFF $ALL$FwCHILDREN
            action: ON  !visible: 1
                do ON $ALL$FwCHILDREN
        state: STANDBY  !color: FwStateOKNotPhysics
20          when ( $ANY$FwCHILDREN in_state ERROR )  move_to ERROR

            when ( $ANY$FwCHILDREN in_state RAMPING )  move_to RAMPING
            when ( $ALL$FwCHILDREN in_state ON )  move_to ON

25          when ( $ANY$FwCHILDREN in_state OFF )  move_to OFF

            action: ON  !visible: 1
                do ON $ALL$FwCHILDREN
            action: OFF !visible: 1
30              do OFF $ALL$FwCHILDREN
            action: STANDBY !visible: 1
                do STANDBY $ALL$FwCHILDREN
        state: ON   !color: FwStateOKPhysics
            when ( $ANY$FwCHILDREN in_state ERROR )  move_to ERROR
35
            when ( $ANY$FwCHILDREN in_state RAMPING)  move_to RAMPING
            when ( $ANY$FwCHILDREN in_state OFF )  move_to OFF

            when ( $ANY$FwCHILDREN in_state STANDBY )  move_to STANDBY
40
            action: STANDBY !visible: 1
                do STANDBY $ALL$FwCHILDREN
            action: OFF !visible: 1
                do OFF $ALL$FwCHILDREN
45          action: ON  !visible: 1
                do ON $ALL$FwCHILDREN
        state: ERROR    !color: FwStateAttention3
            when ( ( $ANY$FwCHILDREN in_state RAMPING ) and
```

```
( $ALL$FwCHILDREN not_in_state ERROR ) )  move_to RAMPING
        when ( ( $ANY$FwCHILDREN in_state OFF ) and
( $ALL$FwCHILDREN not_in_state ERROR ) )  move_to OFF

        when ( $ALL$FwCHILDREN in_state ON )  move_to ON

        when ( ( $ANY$FwCHILDREN in_state STANDBY ) and
( $ALL$FwCHILDREN not_in_state ERROR ) )  move_to STANDBY

        action: ON  !visible: 1
            do ON $ALL$FwCHILDREN
        action: STANDBY !visible: 1
            do STANDBY $ALL$FwCHILDREN
        action: OFF !visible: 1
            do OFF $ALL$FwCHILDREN
    state: RAMPING  !color: FwStateAttention1
        when ( $ANY$FwCHILDREN in_state ERROR )  move_to ERROR
        when ( $ALL$FwCHILDREN in_state ON )  move_to ON
        when ( $ALL$FwCHILDREN in_state STANDBY )  move_to STANDBY
        when ( ( $ALL$FwCHILDREN not_in_state RAMPING ) and
       ( $ANY$FwCHILDREN in_state OFF ) )  move_to OFF
        when ( ( $ALL$FwCHILDREN not_in_state RAMPING ) and
      ( $ANY$FwCHILDREN in_state STANDBY ) )  move_to STANDBY
        action: STANDBY !visible: 1
            do STANDBY $ALL$FwCHILDREN
        action: OFF !visible: 1
            do OFF $ALL$FwCHILDREN
        action: ON  !visible: 1
            do ON $ALL$FwCHILDREN
```

C Wheel translation

```
sort
    Phase = struct WhenPhase ?isWhenPhase | ActionPhase ?isActPhase;
    ActPhaseArgs = struct actArgs(cq: CommandQueue, nrf: IdList, pc: Int, rsc: Bool);
    Id = Nat;
    IdList = List(Id);
    Child = struct child(id:Id, state:State, ptype:PType, busy:Bool);
    Children = List(Child);
    ChildCommand = struct childcommand(id:Id, command:Command);
    CommandQueue = List(ChildCommand);
    PType = struct RPC_Wheel_CLASS ? isRPC_Wheel_CLASS;
    State = struct S_FSM_UNINITIALIZED ? isS_FSM_UNINITIALIZED | S_OFF ? isS_OFF | S_ERROR ? isS_ERROR |
                   S_RAMPING ? isS_RAMPING | S_STANDBY ? isS_STANDBY | S_ON ? isS_ON;
    Command = struct C_STANDBY ? isC_STANDBY | C_OFF ? isC_OFF | C_ON ? isC_ON;

act
    rc,sc,cc: Id # Id # Command;
    rs,ss,cs: Id # Id # State;
    move_state: Id # State;
    move_phase: Id # Phase;
    ignored_command: Id # Command;
    queue_messages: Id;
    enter_then_clause: Id;
    enter_else_clause: Id;
    skip_then_clause: Id;
    start_initialization: Id;
    end_initialization: Id;
    noop_statement: Id;

map
    in_state: Child # State -> Bool;
    in_any_of_states: Child # List(State) -> Bool;
    any_in_state: Children # List(State) -> Bool;
    all_in_state: Children # List(State) -> Bool;
    is_child: Id # Children -> Bool;
    filter_children: Children # PType -> Children;
    filter_children_accu: Children # PType # Children -> Children;
    send_command: Command # Children -> CommandQueue;
    update_state: Id # State # Children -> Children;
    update_busy: Id # Bool # Children -> Children;
    update_busy_all: Bool # Children -> Children;
    remove: Id # IdList -> IdList;
    initialized: Children -> Bool;
    children_to_ids: Children -> IdList;
    busy_children: Children -> Children;
    update_pc: ActPhaseArgs # Int -> ActPhaseArgs;
    reset: ActPhaseArgs -> ActPhaseArgs;

var
    cq: CommandQueue;
```

```
    chs,chs_accu: Children;
    ch: Child;
    id,id1: Id;
    ids: IdList;
    s,s1: State;
    sl: List(State);
    t,t1: PType;
    cmd: Command;
    b, b1, b2: Bool;
    pc, pc1: Int;

eqn
    in_state(child(id,s,t,b),s1) = s == s1;

    in_any_of_states(ch,[]) = false;
    in_any_of_states(ch,s|>sl) = in_state(ch,s) || in_any_of_states(ch,sl);

    any_in_state([], sl) = false;
    any_in_state(ch|> chs, sl) = in_any_of_states(ch,sl) || any_in_state(chs,sl);

    all_in_state([], sl) = true;
    all_in_state(ch|> chs, sl) = in_any_of_states(ch,sl) && all_in_state(chs,sl);

    is_child(id, []) = false;
    is_child(id, child(id1,s,t,b) |> chs) = id == id1 || is_child(id, chs);

    filter_children(chs, t) = filter_children_accu(chs, t, []);

    filter_children_accu([],t,chs_accu) = chs_accu;
    filter_children_accu(child(id,s,t1,b) |> chs, t, chs_accu) =
        if(t==t1,
           filter_children_accu(chs, t, child(id,s,t,b) |> chs_accu),
           filter_children_accu(chs, t, chs_accu));

    send_command(cmd, []) = [];
    send_command(cmd, child(id,s,t,b) |> chs) =
        childcommand(id,cmd) |> send_command(cmd,chs);

    update_state(id, s, []) = [];
    update_state(id, s, child(id1,s1,t,b) |> chs) =
        if(id==id1,
           child(id1,s,t,b) |> chs,
           child(id1,s1,t,b) |> update_state(id,s,chs));

    update_busy(id, b, []) = [];
    update_busy(id, b, child(id1,s,t,b1) |> chs) =
        if(id==id1,
           child(id1,s,t,b) |> chs,
           child(id1,s,t,b1) |> update_busy(id,b,chs));

    update_busy_all(b, []) = [];
    update_busy_all(b, child(id,s,t,b1) |> chs) = child(id,s,t,b) |> update_busy_all(b, chs);

    remove(id, []) = [];
    remove(id, id1 |> ids) =
      if (id == id1,
        ids,
        id1 |> remove(id, ids));

    initialized(chs) = !any_in_state(chs, [S_FSM_UNINITIALIZED]);

    children_to_ids([]) = [];
    children_to_ids(child(id,s,t,b) |> chs) = id |> children_to_ids(chs);

    busy_children([]) = [];
    busy_children(child(id,s,t,true) |> chs) = child(id,s,t,true) |> busy_children(chs);
    busy_children(child(id,s,t,false) |> chs) = busy_children(chs);

    update_pc(actArgs(cq, ids, pc, b), pc1) = actArgs(cq, ids, pc1, b);

    reset(actArgs(cq, ids, pc, b)) = actArgs([], [], 0, b);


proc RPC_Wheel_CLASS(self: Id, parent: Id, s: State, chs: Children, phase: Phase, aArgs: ActPhaseArgs) =
    (
    % BEGIN STATE

    % ------------------
    % BEGIN WHEN CLAUSES

    (
    % BEGIN WHEN
    ((isS_OFF(s)) && (isWhenPhase(phase)) &&
     (any_in_state(chs, [S_ERROR]))) ->
    move_state(self, S_ERROR).
```

```
RPC_Wheel_CLASS(self, parent, S_ERROR, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN
((isS_OFF(s)) && (isWhenPhase(phase)) &&
 (any_in_state(chs, [S_RAMPING]))) ->
move_state(self, S_RAMPING).
RPC_Wheel_CLASS(self, parent, S_RAMPING, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN
((isS_OFF(s)) && (isWhenPhase(phase)) &&
 (all_in_state(chs, [S_STANDBY]))) ->
move_state(self, S_STANDBY).
RPC_Wheel_CLASS(self, parent, S_STANDBY, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN
((isS_OFF(s)) && (isWhenPhase(phase)) &&
 (all_in_state(chs, [S_ON]))) ->
move_state(self, S_ON).
RPC_Wheel_CLASS(self, parent, S_ON, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN
((isS_OFF(s)) && (isWhenPhase(phase)) &&
 ((!(any_in_state(chs, [S_OFF]))) && (any_in_state(chs, [S_STANDBY])))) ->
move_state(self, S_STANDBY).
RPC_Wheel_CLASS(self, parent, S_STANDBY, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN FALLTHROUGH
(((isS_OFF(s)) && (isWhenPhase(phase))) ->
ss(self, parent, s).
move_phase(self, ActionPhase).
RPC_Wheel_CLASS(self, parent, s, chs, ActionPhase, reset(aArgs)))
% END WHEN FALLTHROUGH
))
))
))
))
))
) +
% END WHEN CLAUSES
% ----------------

% --------------------
% BEGIN ACTION CLAUSES
% BEGIN INITIALIZATION CHECK
((isS_OFF(s)) && (isActPhase(phase)) && (!(initialized(chs))) &&
 (pc(aArgs) == 0) && (nrf(aArgs) == [])) ->
    start_initialization(self).
    RPC_Wheel_CLASS(self, parent, s, chs, phase,
              actArgs([], children_to_ids(chs), 0, rsc(aArgs))) <>
% END INITIALIZATION CHECK

% BEGIN CLAUSE SELECTOR
((initialized(chs)) ->
(
(((isS_OFF(s)) && (isActPhase(phase)) && (cq(aArgs) == []) && (pc(aArgs) == 0)) ->
    sum c:Command.(
       rc(parent, self, c).
       (isC_STANDBY(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 1)) <> (
       (isC_OFF(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 2)) <> (
       (isC_ON(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 3)) <> (
       ss(self, parent, s).
ignored_command(self, c).
RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, -1)))))))
)) +
% END CLAUSE SELECTOR

(
% BEGIN ACTION
(((isS_OFF(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->
((
% BEGIN STATEMENT DO
((pc(aArgs) == 1) ->
     queue_messages(self).
     (RPC_Wheel_CLASS(self, parent, s, chs, phase,
               actArgs(send_command(C_STANDBY,
                                    chs), [],  -1, rsc(aArgs)))))
% END STATEMENT DO
))) +
% END ACTION
```

```
((
% BEGIN ACTION
(((isS_OFF(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->
((
% BEGIN STATEMENT DO
((pc(aArgs) == 2) ->
     queue_messages(self).
     (RPC_Wheel_CLASS(self, parent, s, chs, phase,
                actArgs(send_command(C_OFF,
                                     chs), [],  -1, rsc(aArgs)))))
% END STATEMENT DO
))) +
% END ACTION

((
% BEGIN ACTION
(((isS_OFF(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->
((
% BEGIN STATEMENT DO
((pc(aArgs) == 3) ->
     queue_messages(self).
     (RPC_Wheel_CLASS(self, parent, s, chs, phase,
                actArgs(send_command(C_ON,
                                     chs), [],  -1, rsc(aArgs)))))
% END STATEMENT DO
))) +
% END ACTION

(delta))))))
))
% END ACTION CLAUSES
% ------------------

% END STATE
% =========
) +

(
% ===========
% BEGIN STATE

% ------------------
% BEGIN WHEN CLAUSES
(
% BEGIN WHEN
((isS_STANDBY(s)) && (isWhenPhase(phase)) &&
 (any_in_state(chs, [S_ERROR]))) ->
move_state(self, S_ERROR).
RPC_Wheel_CLASS(self, parent, S_ERROR, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN
((isS_STANDBY(s)) && (isWhenPhase(phase)) &&
 (any_in_state(chs, [S_RAMPING]))) ->
move_state(self, S_RAMPING).
RPC_Wheel_CLASS(self, parent, S_RAMPING, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN
((isS_STANDBY(s)) && (isWhenPhase(phase)) &&
 (all_in_state(chs, [S_ON]))) ->
move_state(self, S_ON).
RPC_Wheel_CLASS(self, parent, S_ON, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN
((isS_STANDBY(s)) && (isWhenPhase(phase)) &&
 (any_in_state(chs, [S_OFF]))) ->
move_state(self, S_OFF).
RPC_Wheel_CLASS(self, parent, S_OFF, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN FALLTHROUGH
(((isS_STANDBY(s)) && (isWhenPhase(phase))) ->
ss(self, parent, s).
move_phase(self, ActionPhase).
RPC_Wheel_CLASS(self, parent, s, chs, ActionPhase, reset(aArgs)))
% END WHEN FALLTHROUGH
))
))
))
))
) +
% END WHEN CLAUSES
% ----------------
```

```
% --------------------
% BEGIN ACTION CLAUSES
% BEGIN INITIALIZATION CHECK
((isS_STANDBY(s)) && (isActPhase(phase)) && (!(initialized(chs))) &&
 (pc(aArgs) == 0) && (nrf(aArgs) == [])) ->
    start_initialization(self).
    RPC_Wheel_CLASS(self, parent, s, chs, phase,
              actArgs([], children_to_ids(chs), 0, rsc(aArgs))) <>
% END INITIALIZATION CHECK

% BEGIN CLAUSE SELECTOR
((initialized(chs)) ->
(
(((isS_STANDBY(s)) && (isActPhase(phase)) && (cq(aArgs) == []) && (pc(aArgs) == 0)) ->
    sum c:Command.(
       rc(parent, self, c).
       (isC_ON(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 1)) <> (
       (isC_OFF(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 2)) <> (
       (isC_STANDBY(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 3)) <> (
       ss(self, parent, s).
ignored_command(self, c).
RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, -1)))))))
)) +
% END CLAUSE SELECTOR

(
% BEGIN ACTION
(((isS_STANDBY(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->
((
% BEGIN STATEMENT DO
((pc(aArgs) == 1) ->
     queue_messages(self).
     (RPC_Wheel_CLASS(self, parent, s, chs, phase,
                actArgs(send_command(C_ON,
                                     chs), [],  -1, rsc(aArgs)))))
% END STATEMENT DO
))) +
% END ACTION

((
% BEGIN ACTION
(((isS_STANDBY(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->
((
% BEGIN STATEMENT DO
((pc(aArgs) == 2) ->
     queue_messages(self).
     (RPC_Wheel_CLASS(self, parent, s, chs, phase,
                actArgs(send_command(C_OFF,
                                     chs), [],  -1, rsc(aArgs)))))
% END STATEMENT DO
))) +
% END ACTION

((
% BEGIN ACTION
(((isS_STANDBY(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->
((
% BEGIN STATEMENT DO
((pc(aArgs) == 3) ->
     queue_messages(self).
     (RPC_Wheel_CLASS(self, parent, s, chs, phase,
                actArgs(send_command(C_STANDBY,
                                     chs), [],  -1, rsc(aArgs)))))
% END STATEMENT DO
))) +
% END ACTION

(delta))))))
))
% END ACTION CLAUSES
% ------------------

% END STATE
% ========
) +

(
% ==========
% BEGIN STATE

% ------------------
% BEGIN WHEN CLAUSES
(
% BEGIN WHEN
```

```
((isS_ON(s)) && (isWhenPhase(phase)) &&
 (any_in_state(chs, [S_ERROR]))) ->
move_state(self, S_ERROR).
RPC_Wheel_CLASS(self, parent, S_ERROR, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN
((isS_ON(s)) && (isWhenPhase(phase)) &&
 (any_in_state(chs, [S_RAMPING]))) ->
move_state(self, S_RAMPING).
RPC_Wheel_CLASS(self, parent, S_RAMPING, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN
((isS_ON(s)) && (isWhenPhase(phase)) &&
 (any_in_state(chs, [S_OFF]))) ->
move_state(self, S_OFF).
RPC_Wheel_CLASS(self, parent, S_OFF, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN
((isS_ON(s)) && (isWhenPhase(phase)) &&
 (any_in_state(chs, [S_STANDBY]))) ->
move_state(self, S_STANDBY).
RPC_Wheel_CLASS(self, parent, S_STANDBY, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN FALLTHROUGH
(((isS_ON(s)) && (isWhenPhase(phase))) ->
ss(self, parent, s).
move_phase(self, ActionPhase).
RPC_Wheel_CLASS(self, parent, s, chs, ActionPhase, reset(aArgs)))
% END WHEN FALLTHROUGH
))
))
))
))
) +
% END WHEN CLAUSES
% ----------------

% --------------------
% BEGIN ACTION CLAUSES
% BEGIN INITIALIZATION CHECK
((isS_ON(s)) && (isActPhase(phase)) && (!(initialized(chs))) &&
 (pc(aArgs) == 0) && (nrf(aArgs) == [])) ->
    start_initialization(self).
    RPC_Wheel_CLASS(self, parent, s, chs, phase,
              actArgs([], children_to_ids(chs), 0, rsc(aArgs))) <>
% END INITIALIZATION CHECK
% BEGIN CLAUSE SELECTOR
((initialized(chs)) ->
(
(((isS_ON(s)) && (isActPhase(phase)) && (cq(aArgs) == []) && (pc(aArgs) == 0)) ->
    sum c:Command.(
       rc(parent, self, c).
       (isC_STANDBY(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 1)) <> (
       (isC_OFF(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 2)) <> (
       (isC_ON(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 3)) <> (
       ss(self, parent, s).
ignored_command(self, c).
RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, -1)))))))
)) +
% END CLAUSE SELECTOR
(
% BEGIN ACTION
(((isS_ON(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->
((
% BEGIN STATEMENT DO
((pc(aArgs) == 1) ->
     queue_messages(self).
     (RPC_Wheel_CLASS(self, parent, s, chs, phase,
               actArgs(send_command(C_STANDBY,
                                    chs), [],  -1, rsc(aArgs)))))
% END STATEMENT DO
))) +
% END ACTION

((
% BEGIN ACTION
(((isS_ON(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->
((
% BEGIN STATEMENT DO
((pc(aArgs) == 2) ->
     queue_messages(self).
```

```
      (RPC_Wheel_CLASS(self, parent, s, chs, phase,
                actArgs(send_command(C_OFF,
                                     chs), [],  -1, rsc(aArgs)))))
% END STATEMENT DO
))) +
% END ACTION

((
% BEGIN ACTION
(((isS_ON(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->
((
% BEGIN STATEMENT DO
((pc(aArgs) == 3) ->
     queue_messages(self).
     (RPC_Wheel_CLASS(self, parent, s, chs, phase,
                actArgs(send_command(C_ON,
                                     chs), [],  -1, rsc(aArgs)))))
% END STATEMENT DO
))) +
% END ACTION

(delta))))))
))
% END ACTION CLAUSES
% ------------------

% END STATE
% =========
) +

(
% ===========
% BEGIN STATE

% ------------------
% BEGIN WHEN CLAUSES
(
% BEGIN WHEN
((isS_ERROR(s)) && (isWhenPhase(phase)) &&
 ((any_in_state(chs, [S_RAMPING])) && (!(any_in_state(chs, [S_ERROR]))))) ->
move_state(self, S_RAMPING).
RPC_Wheel_CLASS(self, parent, S_RAMPING, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN
((isS_ERROR(s)) && (isWhenPhase(phase)) &&
 ((any_in_state(chs, [S_OFF])) && (!(any_in_state(chs, [S_ERROR]))))) ->
move_state(self, S_OFF).
RPC_Wheel_CLASS(self, parent, S_OFF, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN
((isS_ERROR(s)) && (isWhenPhase(phase)) &&
 (all_in_state(chs, [S_ON]))) ->
move_state(self, S_ON).
RPC_Wheel_CLASS(self, parent, S_ON, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN
((isS_ERROR(s)) && (isWhenPhase(phase)) &&
 ((any_in_state(chs, [S_STANDBY])) && (!(any_in_state(chs, [S_ERROR]))))) ->
move_state(self, S_STANDBY).
RPC_Wheel_CLASS(self, parent, S_STANDBY, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN FALLTHROUGH
(((isS_ERROR(s)) && (isWhenPhase(phase))) ->
ss(self, parent, s).
move_phase(self, ActionPhase).
RPC_Wheel_CLASS(self, parent, s, chs, ActionPhase, reset(aArgs)))
% END WHEN FALLTHROUGH
))
))
))
))
) +
% END WHEN CLAUSES
% ----------------

% --------------------
% BEGIN ACTION CLAUSES
% BEGIN INITIALIZATION CHECK
((isS_ERROR(s)) && (isActPhase(phase)) && (!(initialized(chs))) &&
 (pc(aArgs) == 0) && (nrf(aArgs) == [])) ->
    start_initialization(self).
```

```
    RPC_Wheel_CLASS(self, parent, s, chs, phase,
              actArgs([], children_to_ids(chs), 0, rsc(aArgs))) <>
% END INITIALIZATION CHECK
% BEGIN CLAUSE SELECTOR
((initialized(chs)) ->
(
(((isS_ERROR(s)) && (isActPhase(phase)) && (cq(aArgs) == []) && (pc(aArgs) == 0)) ->
    sum c:Command.(
       rc(parent, self, c).
       (isC_ON(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 1)) <> (
       (isC_STANDBY(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 2)) <> (
       (isC_OFF(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 3)) <> (
       ss(self, parent, s).
ignored_command(self, c).
RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, -1)))))))
)) +
% END CLAUSE SELECTOR
(
% BEGIN ACTION
(((isS_ERROR(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->
((
% BEGIN STATEMENT DO
((pc(aArgs) == 1) ->
     queue_messages(self).
     (RPC_Wheel_CLASS(self, parent, s, chs, phase,
              actArgs(send_command(C_ON,
                                   chs), [],  -1, rsc(aArgs)))))
% END STATEMENT DO
))) +
% END ACTION
((
% BEGIN ACTION
(((isS_ERROR(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->
((
% BEGIN STATEMENT DO
((pc(aArgs) == 2) ->
     queue_messages(self).
     (RPC_Wheel_CLASS(self, parent, s, chs, phase,
              actArgs(send_command(C_STANDBY,
                                   chs), [],  -1, rsc(aArgs)))))
% END STATEMENT DO
))) +
% END ACTION
((
% BEGIN ACTION
(((isS_ERROR(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->
((
% BEGIN STATEMENT DO
((pc(aArgs) == 3) ->
     queue_messages(self).
     (RPC_Wheel_CLASS(self, parent, s, chs, phase,
              actArgs(send_command(C_OFF,
                                   chs), [],  -1, rsc(aArgs)))))
% END STATEMENT DO
))) +
% END ACTION
(delta))))))
))
% END ACTION CLAUSES
% ------------------

% END STATE
% =========
) +

(
% ===========
% BEGIN STATE

% ------------------
% BEGIN WHEN CLAUSES
(
% BEGIN WHEN
((isS_RAMPING(s)) && (isWhenPhase(phase)) &&
 (any_in_state(chs, [S_ERROR]))) ->
move_state(self, S_ERROR).
RPC_Wheel_CLASS(self, parent, S_ERROR, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN
((isS_RAMPING(s)) && (isWhenPhase(phase)) &&
 (all_in_state(chs, [S_ON]))) ->
move_state(self, S_ON).
RPC_Wheel_CLASS(self, parent, S_ON, chs, phase, aArgs) <>
% END WHEN
```

```
((
% BEGIN WHEN
((isS_RAMPING(s)) && (isWhenPhase(phase)) &&
 (all_in_state(chs, [S_STANDBY]))) ->
move_state(self, S_STANDBY).
RPC_Wheel_CLASS(self, parent, S_STANDBY, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN
((isS_RAMPING(s)) && (isWhenPhase(phase)) &&
 ((!(any_in_state(chs, [S_RAMPING]))) && (any_in_state(chs, [S_OFF])))) ->
move_state(self, S_OFF).
RPC_Wheel_CLASS(self, parent, S_OFF, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN
((isS_RAMPING(s)) && (isWhenPhase(phase)) &&
 ((!(any_in_state(chs, [S_RAMPING]))) && (any_in_state(chs, [S_STANDBY])))) ->
move_state(self, S_STANDBY).
RPC_Wheel_CLASS(self, parent, S_STANDBY, chs, phase, aArgs) <>
% END WHEN
((
% BEGIN WHEN FALLTHROUGH
(((isS_RAMPING(s)) && (isWhenPhase(phase))) ->
ss(self, parent, s).
move_phase(self, ActionPhase).
RPC_Wheel_CLASS(self, parent, s, chs, ActionPhase, reset(aArgs)))
% END WHEN FALLTHROUGH
))
))
))
))
))
) +

% END WHEN CLAUSES
% ---------------

% --------------------
% BEGIN ACTION CLAUSES
% BEGIN INITIALIZATION CHECK
((isS_RAMPING(s)) && (isActPhase(phase)) && (!(initialized(chs))) &&
 (pc(aArgs) == 0) && (nrf(aArgs) == [])) ->
    start_initialization(self).
    RPC_Wheel_CLASS(self, parent, s, chs, phase,
                actArgs([], children_to_ids(chs), 0, rsc(aArgs))) <>
% END INITIALIZATION CHECK
% BEGIN CLAUSE SELECTOR
((initialized(chs)) ->
(
(((isS_RAMPING(s)) && (isActPhase(phase)) && (cq(aArgs) == []) && (pc(aArgs) == 0)) ->
    sum c:Command.(
       rc(parent, self, c).
       (isC_STANDBY(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 1)) <> (
       (isC_OFF(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 2)) <> (
       (isC_ON(c) -> RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, 3)) <> (
       ss(self, parent, s).
ignored_command(self, c).
RPC_Wheel_CLASS(self, parent, s, chs, phase, update_pc(aArgs, -1)))))))
)) +
% END CLAUSE SELECTOR

(
% BEGIN ACTION
(((isS_RAMPING(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->
((
% BEGIN STATEMENT DO
((pc(aArgs) == 1) ->
     queue_messages(self).
     (RPC_Wheel_CLASS(self, parent, s, chs, phase,
                actArgs(send_command(C_STANDBY,
                                     chs), [],  -1, rsc(aArgs)))))
% END STATEMENT DO
))) +
% END ACTION

((
% BEGIN ACTION
(((isS_RAMPING(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->
((
% BEGIN STATEMENT DO
((pc(aArgs) == 2) ->
     queue_messages(self).
     (RPC_Wheel_CLASS(self, parent, s, chs, phase,
                actArgs(send_command(C_OFF,
```

```
                                    chs), [],  -1, rsc(aArgs)))))
    % END STATEMENT DO
    ))) +
    % END ACTION

    ((
    % BEGIN ACTION
    (((isS_RAMPING(s)) && (isActPhase(phase)) && (cq(aArgs) == [])) ->
    ((
    % BEGIN STATEMENT DO
    ((pc(aArgs) == 3) ->
         queue_messages(self).
         (RPC_Wheel_CLASS(self, parent, s, chs, phase,
                    actArgs(send_command(C_ON,
                                    chs), [],  -1, rsc(aArgs)))))
    % END STATEMENT DO
    ))) +
    % END ACTION

    (delta))))))
    ))
    % END ACTION CLAUSES
    % -----------------

    % END STATE
    % =========
    ) +

    (
    % BEGIN GENERIC CLAUSES (shared by all states)
    sum id:Id.(sum s1:State.(((isActPhase(phase)) && (is_child(id, chs)) &&
                              (pc(aArgs) == 0) && (initialized(chs))) ->
       rs(id, self, s1).
       move_phase(self, WhenPhase).
       RPC_Wheel_CLASS(self, parent, s, update_busy(id, false, update_state(id, s1, chs)), WhenPhase, reset(aArgs)))) +

    sum id:Id.(sum s1:State.(((isActPhase(phase)) && (is_child(id, chs)) &&
           ((pc(aArgs) > 0) ||
            ((pc(aArgs) == -1) && (cq(aArgs) != [])))) ->
       rs(id, self, s1).
       RPC_Wheel_CLASS(self, parent, s,
                    update_busy(id,
                                false,
                                update_state(id, s1, chs)),
                    phase, aArgs))) +

    ((isActPhase(phase)) && (cq(aArgs) != []) && (!(initialized(chs)))) ->
        sc(self, id(head(cq(aArgs))), command(head(cq(aArgs)))).
            RPC_Wheel_CLASS(self, parent, s,
                            update_busy(id(head(cq(aArgs))), true, chs),
                            phase,
                            actArgs(tail(cq(aArgs)),
                                    (id(head(cq(aArgs))))|>(nrf(aArgs)), pc(aArgs), rsc(aArgs))) +

    ((isActPhase(phase)) && (cq(aArgs) != []) && initialized(chs)) ->
        sc(self, id(head(cq(aArgs))), command(head(cq(aArgs)))).
            RPC_Wheel_CLASS(self, parent, s,
                            update_busy(id(head(cq(aArgs))), true, chs),
                            phase,
                            actArgs(tail(cq(aArgs)), [], pc(aArgs), rsc(aArgs))) +

    sum id:Id.(sum s1:State.(
           ((isActPhase(phase)) && (cq(aArgs) == []) && (nrf(aArgs) != []) && (is_child(id, chs)) &&
            (!(initialized(chs)))) ->
                  rs(id, self, s1).
                  ((initialized(update_state(id,s1,chs))) ->
                       end_initialization(self).
                       RPC_Wheel_CLASS(self, parent, s, update_state(id,s1,chs), phase,
                                       actArgs(cq(aArgs), remove(id, nrf(aArgs)), -1, rsc(aArgs)
                                       )) <>
                       RPC_Wheel_CLASS(self, parent, s, update_state(id,s1,chs), phase,
                                       actArgs(cq(aArgs), remove(id, nrf(aArgs)), -1, rsc(aArgs)
                                       ))))) +

    ((isActPhase(phase)) && (cq(aArgs) == []) && (initialized(chs)) && (pc(aArgs) == -1)) ->
        move_phase(self, WhenPhase).
        RPC_Wheel_CLASS(self, parent, s, chs, WhenPhase, reset(aArgs))

    % END GENERIC CLAUSES
    );

init
    allow({cs, cc, move_state, move_phase, ignored_command,
           queue_messages, enter_then_clause, enter_else_clause,
           skip_then_clause, start_initialization, end_initialization,
```

```
815         noop_statement},
        comm({rs|ss -> cs, rc|sc -> cc},
        RPC_Wheel_CLASS(1, 1, S_OFF, [],
                        ActionPhase,actArgs([], [], 0, false))));
```